On the Creation of MipFlex. Extending JuMP and Aiding Solvers With Custom Recognition Procedures.

Henriette Andersen

Magnus Lie Hetland (Supervisor)

February 2023

Abstract

Many feasibility and optimization problems can be expressed as linear programs when the decision variables form linear constraints and objective functions. Integer programs; a subclass of problems where one or more decision variables have to be integer, is appearing a great deal in various industries. Examples of use are within production planning, where one cannot produce fractional wares, or within scheduling, where each variable is required to be binary, and, for example, corresponds to assigning a vehicle to a route or not. Integer programming is NP-Complete, and it is therefore made many optimization solvers, both commercial and open source, for use in industries. For the commercial ones it is hard to know the exact methods in use, but for the open source we see that there often are similar procedures in use, applied to every problem. By crafting procedures of recognition, we hope to find that more tailored algorithms can speed up the process of solving specific problem classes. In this research we will therefore explore how one can make a seamless layer of recognition procedures for deciding which algorithms to use, while still being able to use a solver for problems not recognized by this system. For interacting with solvers one may use an algebraic modeling language (AML) in order to formulate the problems at a higher level, and we argue that a plugin at this level will be able to make the problem solving as fast, or faster on certain problem domains, as an open source solver. We will produce the prototype MipFlex, which makes use of Julia's AML JuMP. We will explore how the layout can be flexible and extendable, and compare how the prototype effects execution time over different problem classes and solvers. We will also dive into a specific recognition procedure in order to have a specific use case for MipFlex.

Keywords— MipFlex, mixed integer programming, optimization solvers, JuMP, recognition procedures.

For my grandmother, Wenche Andersen,
who always supported and believed in me, and taught me to never give up.

Acknowledgements

I would like to thank my amazing supervisor, Magnus Lie Hetland. Without his superb advice, both on this project, but also on how to handle the slightly unsteady life of writing a thesis, this would probably not have succeeded. Further I would thank Halvard Hummel for taking the time to look at some of the rather problematic parts of the proofs, and coming with more insight. Lastly I would like to thank my partner in crime, and in life, Per Kristian Ørke. Your cheering, hugs, and prepared luxury dinners, made all the difference.

Chapter 1

Introduction

The main focus of this work is to explore the possibility and results of implementing a module which provides the possibility of inserting recognition and solving procedures of different types of linear programs, while still being able to connect existing linear programming solvers to the module, which can work simultaneously on the problem, or take main control if the problem class is not recognizable by the specific procedures. In this section we discuss why and how this could be helpful within different fields of work, and what already exists for working with these kind of problems.

Linear programs (LP's) appear when one considers problems where the variables form linear constraints, and possibly containing a linear objective function to maximize or minimize. These types of problems appear in a lot of different domains, and examples are graphical problems, such as network and flow problems within transportation of goods, optimization within microeconomics, decision problems connected to time slot management, and production scheduling, for example, within hydro-power[5]. Obviously it has a lot of use-cases, and it is to no surprise that there exist several commercial and properly maintained solvers for linear programming, such as Gurobi[15] and CPLEX[11], for use in different businesses. There are also several open source solvers, such as HiGHS[1], GLPK[21], Cbc[9], and SCIP[7].

Often one would need to constrain some or all of the solution variables to be integers. Such mixed integer linear problems (MILP's) are proven to be NP-hard, meaning that there exists no polynomial-time algorithm solving the general problem, unless P=NP. However, there exist different heuristics that can improve the solution time if one know certain information from before, and different solvers have settings to help with heuristics. Gurobi support for feasibility heuristics, and is arguably one of the solvers with most customization possibilities. There exists several general MIP heuristics[6]. The open source solver SCIP has implemented a lot of such heuristics, and is one of the fastest open-source solver. However, these open source solvers does not have ways of entering custom procedures that can be run whenever it recognizes a specific problem, or send more detailed information about the nature of the problem, which an algorithm could use beneficially. Instead the use of general algorithms, which are exponential in the number of variables and constraints for integer programs, will be used in the solvers, where one could instead have used well-known graph algorithms of polynomial complexity.

Our aim is to implement an open source module which can be integrated with solvers. Thus one have the possibility to search for known sub-classes of problems, or send in data about a problem class already known, which can then be used to apply specific algorithms instead of generic branch-and-bound type solver algorithms. We aim to make it extendable, so that further recognition and solver procedures can be added by need, and flexible, so that one can choose which recognition procedures to run. We will then test if it can give any improvement compared to existing open source solvers, and show that in certain cases it can be quite useful. We finish by exploring some ideas of additional implementation which could make this module even more useful.

Chapter 2

Background and Preliminaries

In this chapter we will introduce some basic concept of integer programming, and show some examples of how it is used in order to illustrate how important it is in industries. We will also examine existing technologies when it comes to representation and solving of linear programs, and discuss how we can use these to possibly extend the flexibility of open source optimization.

2.1 Integer Programming

Linear programs can be expressed in several ways, but a maximization problem on standard form is given as

$$\begin{aligned} \max_{\mathbf{x}\in\mathbb{R}^n} \quad & \mathbf{c}^T\mathbf{x}, \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x}. \geq \mathbf{0}, \end{aligned} \tag{2.1}$$

where $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m\times n}$, $\mathbf{b} \in \mathbb{R}^m$, and $\mathbf{0}$ is an n-dimensional vector of only zeros. The last inequality is meant to be taken element-wise, so that each element of $\mathbf{x}$ should be greater than zero. So there is m linear constraints defined by $\mathbf{A}$ and $\mathbf{b}$, plus the n constraints of non-negativity, and we want to find an

$\mathbf{x}$ of real numbers such that no constraints are violated, and, if several such solutions exists, take the one which maximizes the objective function.

Integer programming (IP) is a special case of LP, where all variables are limited to only take on integer values. If only some of the variables are constrained to integer values, the problem is called a mixed integer program (MIP). There are a lot of interesting results regarding linear programming, especially connected to its dual and the number of solutions, which can be used in different solution procedures, however, the scope for this thesis does not include theory connected to duality of LP's.

An example of the use of integer programming is the transportation problem. Assume we have a company producing indivisible goods, such as cinnamon buns. Every day the n different bakeries have different production capacities given by $b_1, b_2, \dots b_n$, while the m grocery stores selling them have different demands given by $g_1, g_2, \dots g_m$. Since the distance between different supply and demand venues differs, it is estimated a cost of transport, given as cost per cinnamon bun on a given route, between pairs of bakeries and grocery stores. If there is no viable transportation route between two venues, then no cost is calculated and it is considered impossible to transfer goods between these units. This makes up a directed bipartite graph.

This problem can be modeled as a linear problem as follows. Consider the graph $\mathbf{G}$ with vertex set $\mathbf{V} = \mathbf{B} \bigcup \mathbf{G}$ of size $n + m$, where $\mathbf{B}$ and $\mathbf{G}$ are the disjoint sets of bakeries and grocery stores. Let the edge set $\mathbf{E}$ be every directed viable route from a bakery to a grocery store, and let the cost of these routes be denoted c_{ij}, where $i \in \mathbf{B}$ and $j \in \mathbf{G}$. Let f_{ij} denote the transportation of buns per day from bakery i to grocery store j. The problem becomes to meet the demand with the available resources while minimizing cost:

$$
\begin{aligned}
\min \quad & \sum_{(i,j)\in\mathbf{E}} c_{ij} f_{ij}, \\
\text{subject to} \quad & \sum_{\{j|(i,j)\in\mathbf{E}\}} f_{i,j} \leq b_i, \quad \forall i \in \mathbf{B} \\
- & \sum_{\{i|(i,j)\in\mathbf{E}\}} f_{i,j} \leq -g_j, \quad \forall j \in \mathbf{G} \\
& f_{ij} \geq 0 \quad \wedge \quad f_{ij} \in \mathbb{Z}, \quad \forall (i,j) \in \mathbf{E}.
\end{aligned}
$$

If the transport costs and the production capacity of bakeries are both integer values, then it can be shown that basic feasible solutions are integer. Basic feasible solutions (BFS's) are feasible solutions with a minimal set of non-zero variables, and the famous Simplex[12] algorithm searches for optimal solutions by moving from BFS to BFS. However, if the transport costs are fractional, it may not find a default integer solution, and one may need to cut the solution space in half at several points and make a search tree for an integer solution. This can take exponential time. However, the family of problems and methods used can impact the running time a lot, thus using heuristic and knowledge about the problem can be crucial.

Another example is within decision problems, where the variables are restricted to take on either 1 or 0. For a simplified example for financing projects, let there be three possible projects a company can take on, and all will take two years. Each year the cost of each project is estimated, and the available capital is given. The gain in revenue after completion is also estimated per project. The data could look something like table 2.1.

		Year 1	Year 2
Project	Gain	Cost	Cost
1	0.3	0.2	0.4
2	0.2	0.1	0.3
3	0.23	0.12	0.3
Available capital		0.35	0.65

Table 2.1: Example data connected to an decision problem.

If we let x_i represent project number i, where $i \in [1,3]$, we can make a representation of the problem, where a solution with $x_i = 0$ means we do not take on project i, while we take it on if $x_i = 1$. In this example we can only finance the projects with the yearly available capital, and we want to find the combination of affordable projects which will maximize profit.

$$\begin{aligned} \text{Maximize} \quad & 0.3x_1 + 0.2x_2 + 0.23x_3 \\ \text{Subject to} \quad & \\ & 0.2x_1 + 0.1x_2 + 0.12x_3 \leq 0.35 \\ & 0.4x_1 + 0.3x_2 + 0.3x_3 \leq 0.65 \\ \text{where} \quad & x_i \in \{0,1\} \forall i. \end{aligned}$$

For this example, we can simply check all the 2^3 possibilities, list which are feasible, calculate the objective value for those, then choose the optimal solution. However, when the list of variables increase, this will take exponential time, so a faster algorithm is needed.

It is shown that LP's can be solved in strongly polynomial time[24], whereas IP's can be proved to be NP-hard by a reduction from the minimum vertex cover problem. Since IP is a special case of MIP, MIP is at least as hard as IP, and so is also NP-hard. Binary linear optimization problems, as the example from table 2.1, are specializations of IP, and some of these can be solved strongly polynomial, while some cannot[22]. As such it could be beneficial to recognize instances where there is a strong polynomial algorithm, or very special instances of IP or MIP where it is known to be easier approaches.

2.2 Algorithmic Modeling Languages

An optimization solver accepts input on specific formats, and often it is neither easy to read for a human, nor compact. Algorithmic Modeling Languages (AML's) are helpful tools when dealing with optimization solvers, as they allow for problems to be written in a logical mathematical way for the user, and translate it into accepted input for the solver. Some AML's are made independently with their own syntax, like the commercial AMPL and GAMS, while others are embedded in their own programming languages, such as YALMIP and CVX for Matlab, and PuLP and Pyomo for Python, which makes it easy for users to manipulate and create different models from data directly within the language, and use the results in their application. However, embedded AML's have to follow the rules of its native language's syntax and grammar, or find workarounds, which can be a challenge for making it expressive and easy to use.

For storing a sparse matrix, a common method used for when the model is being built is to store two arrays representing each row. One array specifies the indices of the non-zero variables, and the other stores the value of these respectively. AML's in interpreted languages often use operator overloading to achieve this, overloading the basic operators on variables, building the final expression by combining sub-expressions, which can result in many intermediate memory-allocations. AMPL is quite effective memory-wise as it statically determines how many elements it needs, typically by passing through the expression twice, determining the final storage required at the first pass, then determining the values to store at the second pass. JuMP[13] tries to mimic the latter approach through metaprogramming using syntactic macros, which substitute elements of the syntax tree, instead of lexical tokens as done for macros in C. The macros are evaluated only once at compile time, possibly within execution time by use of just-in-time compilation, so it introduces no extra runtime overhead. JuMP can also utilize multiple dispatch, a feature of Julia, so that no matter the type of variable received, there is a function defined for it. These macros make it possible for significantly fewer memory allocations than operator overloading does, and in fact, the authors of JuMP show that JuMP performs on the same order of magnitude as commercial AML's, while overloading AML's built on Python are slower by a factor of 10. These results were measured from the time used to build in-memory models and reporting them as MPS (Mathematical Programming Structure) and LP formats, all performed on the same machine.

From the promising benchmarks of JuMP's performance, it seems logical to use Julia's JuMP for our AML, as it is open source, intuitive to use, and opens up for constructing a plugin within the same native language.

In 2019 JuMP was completely rewritten, intorducing MathOptInterface[17] (MOI), in order to support the big diversification in expected data structures from the solvers. The problem JuMP had taken on, of translating LP's or non-linear programs (NLP's) into a *standard form* for any solver to use, had gotten much harder because of the widening of the field in general, and the many independent solvers available. Not only are they different in the intended problems to solve, e.g. MIP, NLP, conic programs, etc., there are also solvers for the same type of problems, expecting the input in different formats, even though the logical representations are equivalent.

For example, some solvers may want the constraints in a less-than manner as we saw in eq. (2.1), while others may want it with equality, which is the starting point for most Simplex algorithms. Achieving equality from an inequality is a matter of introducing slack variables, so that

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b$$

becomes

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n + s = b \quad \wedge \quad s \geq 0,$$

where the a's are real constants, and the $x's$ and the s are real variables.

From a greater-than inequality, one can either convert it to a less-than inequality by multiplying by -1 on both sides, and then add the slack variable. Alternatively one can directly add a surplus variable, which should be less than zero, or subtract a surplus variable which should be greater than zero. Some solvers may want a vector containing elements representing $\leq$, $\geq$ and $=$, in order to identify the bounds of the constraints, while others again may want both a lower and upper bound for each constraint.

In order to solve this problem of high variety in solver standards, the JuMP developers decided to rewrite the system to be more flexible, not just from the solver side, but also from the user side. If a user could to write a problem in less-than inequalities, and additionally be able to send it to two different solvers requiring greater-than and qualities respectively, it would benefit the user greatly. The result of this rewriting was the introduction of an intermediate abstract data structure they named `MathOptInterface` (MOI).

MOI is made to hold mathematical optimization data in a general way, and easy to interpret. The idea is that every constraint is represented as a `Function-in-Set` pair. More information on which functions and sets are supported can be found in the documentation.

Another highly convenient feature, the idea that made the new implementation of JuMP so flexible, is the rewriting system called *bridges*. Basically bridges translates problems into equivalent forms, so that a user can send in the problem, and then bridges will be applied in order to try and translate the problem into a version acceptable by the solver to be used. The solver developers may specify which bridges can be used.

If the problem is big, and there are many different forms of the problem, it may take additional overhead to use bridges. In [17] they offer benchmarks showing that the runtime increases by a factor of 4 for use with GPLK, and 4-7 for SCS. But one have the choice to bypass bridging if one wish to speed things up, and then the overhead is around a factor of 1-2.5 compared to not using MOI, which is almost nothing at all.

MOI introduced `MathOptFormat`, a file format making it easy to store and read big collections of data. As part of this file format a concrete description of MOI was provided as a JSON schema, which serializes MOI models. It is a concrete description of the functions and sets of MOI, and serves as a canonical representation that may be updated iterative as more functions and sets are supported.

2.3 Linear Programming Solvers

There are a number of available solvers aimed at solving optimization programs, directed at different types of problems. For example, some aim at general linear or non-linear problems, whereas some are specified to work on specific forms of non-linear programs, such as quadratic programs or second-order conic programs. We are focusing on the solvers intended for linear programming with mixed integer constraints, which we can divide into commercial and open source solvers. Among the commercial ones, we find solvers like Gurobi, CPLEX, MOSEK and FICO Xpress. Since commercial solvers are highly optimized while not being very transparent regarding their exact algorithms, we focus more on the open source MIP solvers with support for JuMP. If we can help enhancing the use of those, it will be a good start, and highly practical for use in open source projects. The open source solvers with API suport for JuMP are currently GLPK (GNU Linear Programming Kit), HiGHS (Linear optimization software), Cbc (COIN-OR Branch and Cut), and SCIP (Solving Constraint Integer Programs).

The branch-and-cut algorithm [2] is a fairly common method for most MIP-solvers, and we present the workings of the one used by GLPK. The idea is to maintain a current bound on the objective function value, then search for better ones by making a search tree. This idea by itself is called branch-and-bound. By introducing cutting planes, as will be described below, we get the branch-and-cut method.

The algorithm is given in pseudocode in algorithm 1. Note that the indentations here marks when a statement ends.

Algorithm 1 Branch-and-Cut

BRANCH-AND-CUT($MIP, Lazy, Cuts$)

1: $z^{best} := +\infty$
2: $\mathbf{x}^{best} := \mathbf{0}$
3: add the pair (MIP, z^{best}) to list L
4: **while** L not empty
5: select a pair (P, z) from L
6: use known LP-algorithm to solve P without integer constraints,
7: and obtain solution $\mathbf{x}$ with objective value z^P from this
8: **if** P infeasible or $z^P \geq z^{best}$
9: remove (P, z) from L
10: **continue**
11: **if** any set of constraints $C \subset Lazy$ violates solution
12: extend P with C
13: **goto 6**
14: **if** $\mathbf{x}$ is an integer solution
15: **if** solution is unbounded
16: **return** solution $\mathbf{x}, z^P,$ "*unbounded*"
17: $z^{best} := z^P$
18: $\mathbf{x}^{best} := \mathbf{x}$
19: remove any pairs $(P, z) \in L$ with $z \geq z^{best}$
20: **continue**
21: **if** any set of constraints $C \subset Cuts$ violates solution
22: extend P with C
23: **goto 6**
24: take a fractional $x \in \mathbf{x}$ and let x^* be its fractional value
25: let $P_1 = P \bigcup (x \leq \lfloor x^* \rfloor)$
26: let $P_2 = P \bigcup (x \geq \lceil x^* \rceil)$
27: add (P_1, z^P) and (P_2, z^P) to L
28: remove (P, z) from L
29: **if** $z^{best} \neq +\infty$
30: **return** optimal solution $\mathbf{x}^{best}, z^{best},$ "*optimal*"
31: **else**
32: **return** "$infeasible$"

Assume that we work with a minimization problem. One starts by setting the best known solution to infinity, $z^{best} = +\infty$, then adds the LP program to a list, L, of active search nodes.

The algorithm selects a program from the problem set L. If there are none, the algorithm terminates and will check what the current best bound is. If there is a program in L, the algorithm then relaxes the integer constraint, solving the relaxed LP, most often with the use of the Simplex method or the interior-point method. Simplex, developed by Dantzig, is known for being very efficient in practice, but its worst case runtime is exponential. It often uses many relatively small cost iterations, where it searches vertices of the polyhedron the constraints make up, as the constraints in theory are convex cutting planes in a $\mathbb{R}^n$ space, where n is the number of variables. The interior-point-method is an approximation algorithm based on Newton's method, where one typically adds a logarithmic barrier function to the objective function. Since the first and second derivatives are required for this method, each iteration can be more expensive than for the Simplex method, but it typically converges faster to an answer.

If the objective value of the relaxed LP, z^P, is worse than z_{best}, in minimization meaning $z^P \geq z_{best}$, then the current branch of the search tree is removed from the node set, and a new problem from the list is selected, if possible. The same happens if there are no feasible solutions to the relaxed LP. However, if there is a z^P with $z^P \leq z_{best}$, the algorithm continues to the next step.

A user may mark constraints in the original problem as *lazy*, meaning that they don't think the constraints will restrict the optimal solution, but it still needs to be fulfilled. An example could be that you never have more storing capacity than one million units, but you are fairly sure you would never have material enough to produce more than a hundred thousand units either way. You still need to fulfill the requirement, but it would likely not affect the result. Constraints marked as lazy will be removed from the initial problem of the procedure, and will be stored in a *lazy* pool. If, at a point after finding a relaxed solution, the algorithm detects that any of those lazy constraints violate the solution, they are added back in so that the relaxed solution can be better estimated, and possible integer solutions violating the lazy constraint can be ruled out.

When a relaxed solution that do not violate any lazy constraints is found, a check for integrality is made, checking if the optimal solution to the relaxed version also is optimal for the integer version. If it is an integer feasible solution, the best upper bound is updated, $z^{best} := z^L$, together with what the variables are set to in this solution. Then branches with $z^P \geq z_{best}$ are pruned from the search tree, including the current search branch, since no better solution can be found in this search space, and goes back to the set of problems, L, to search further.

If any variable in a solution is fractional, there are possibilities for adding cutting planes. Cutting planes are constraints that make the relaxed solution invalid without removing any feasible integer solutions, thereby *forcing* the relaxed version to give an even better estimate next time it is solved. The user can set options for which kind of cuts to be generated, but common for them is that they aim at cutting away as much of the solution space as possible without removing feasible integer solutions.

After resolving the relaxed LP with cutting planes, and again checking for integrality but not obtaining it, one chooses a fractional variable to branch upon. If x is the chosen variable with value x^* at the relaxed solution, one creates two sub-problems, where one has the constraint $x \leq \lfloor x^* \rfloor$, and the other has $x \geq \lceil x^* \rceil$. By doing this we know that if there exist an optimal solution to the problem, one of the sub-problems will contain it, as no feasible solution will be ruled out. These problems are added to the set L, together with the current best solution to the fractional problem in this branch, while the current problem is removed from the set. Then one chooses a new problem of L to search further.

When the list of active programs, L, is empty, one checks what z^{best} is set to. If it still is $+inf$ it means that the original problem has no feasible solution. Otherwise the last stored integer feasible solution is an optimal one. If the objective function is unbounded from below, a LP-relaxation will be unbounded, and so the algorithm can terminate and report this if it happens.

GLPK offers the user to add specific callback routines which can be used to tailor different behavior of the solver if wanted. The callbacks are connected to different event flags, and are mostly meant to tweak the current algorithm, not to use any specific recognition procedures. Examples of callback routines are requests for sub-problem selection when choosing a new one from L, request

for prepossessing of the problem, request for heuristic approaches to reach an integer solution from a fractional solution, request for cut generation, or for which variable to branch upon. Many of these events can also be set to be handled by predefined procedures beforehand. For example, backtracking using *the best projection* heuristic is set to default for choosing new problems from L, but it can be switched to depth or breadth first through an option. Similarly, common cut generators, such as Gomory cuts or MIR (mixed integer rounding) cuts, can be set to be used.

Cbc (COIN-OR branch-and-cut) is also, as the name suggests, a branch-and-cut solver. It can make use of the cut-generation library Cgl, also a COIN-OR project[18].

SCIP advertises on their web page that it is "currently one of the fastest non-commercial solvers for mixed integer programming and mixed nonlinear programming, and it uses branch-cut-and-price. Branch-and-price is a special case of branch-and-bound where the relaxation is done by the column generation method. This method considers a subset of the original variables, hoping that a sub-problem can find good bounds to the solution. Further, a *pricing* function is formed with the current solutions to the sub-problem and its dual. It is formed such that minimizing this function over the unused variables (in case of a minimization problem) will give variables that can improve the bound if included in the sub-problem. This, together with cuts in the branch-cut-and-price method, is used to improve the bounds.

HiGHS is the last open source MIP solver which has support for JuMP, and it uses branch-and-bound, so we see that all these solvers use more or less a variation of the same method.

2.3.1 Previous Benchmarking

In order to help with research on real-world LP's and MIP's, a benchmarking library, MIPLIB[14], was made by Robert E. Bixby, E.A. Boyd, and R.R. Indovina in 1992. The most recent update to the library was in 2017, where they received 5,721 submissions from different partners in the industry. From these initial submissions they created two subsets; a small benchmark set, and a bigger collection. The benchmark set was narrowed down to only 240 instances via a number of processes, one of which being that they measured different features of the problems and tried to choose problems in order to cover a large spectrum of each feature. In addition, each instance in the benchmark set was solvable by the union of available code at time of creation, and were also chosen in a way that make them easier for testing with regards to numerical stability.

H. Mittelmann[16] has regularly tested MIPLIB against current commercial and open source solvers. His results from MIPLIB2017[19] is summarized on his web page when running the code with a "limit of 2 hours on an Intel i7-11700K, 8 cores and 8 threads, 64GB, 3.6Ghz". Figure 2.1[20] presents the mean of these run times. It does not include GLPK, but all the other MIP solvers we have looked at, in addition to the commercial solvers Gurobi, COPT and SCIP with CPLEX, are included.

Unscaled and scaled shifted geometric means of run times

All non-successes are counted as max-time.
The third line lists the number of problems (240 total) solved.

	CBC	Gurobi	COPT	SCIP	SCIPC	HiGHS
unscal	1328	81.5	214	888	727	756
scaled	16.3	1	2.63	10.9	8.92	9.27
solved	107	227	197	137	152	156

Figure 2.1: Mittelmann's benchmarks of MIPLIB2017.

From this we can see that the commercial solvers with no doubt outperforms the open source ones. The number of unsolved instances will of course increase run time as the max-time then is used. However, if one inspects the table[19] of runtimes for when 8 threads is used, one can see that whenever all solvers solve the problem, the commercial solvers are generally much faster. As we do not know what kind of recognition procedures these commercial solver may have, it is not that constructive to compare a new structure of recognition procedures against these. Thus, as mentioned, we stick to looking at the open-source solvers.

2.3.2 Preprocessing and Presolving

Presolving techniques are generally techniques to improve the representation of a MIP in some fashion. This could be to tighten bounds, remove redundant bounds, or recognize easy infeasibility criteria. There are several texts on this issue (e.g. [23]), and most solvers have some of these by default, while others can be chosen.

GLPK can for example tighten bounds of some variables, or remove bounds of some redundant constraints during preprocessing. SCIP has it connected presolving library, PaPILO, which is used as default since version 7.0 of SCIP. PaPILO can also be linked to, among others, HiGHS or Gurobi, and it provides parallel presolve routines for LP's and MIP's.

How solvers do prepossessing of a problem could be helpful to look at, in that if we do the same in our module, we start at the same ground as any solver with preprocessing capabilities. However, one can also argue that if the solvers do this, why do the same work in parallel? Unless preprocessing could help the recognition procedures drastically, we could let the solvers handle it, and let our module implement unused methods instead. Exploring how different preprocessing methods would impact the time taken by recognition and solving procedures could be interesting to look at in a later iteration of the module. An optional preprocessing feature could for example be implemented.

We see from the creation of MIPLIB2017 that they skip most presolving when providing the benchmark set, as this is one aspect to test the solvers on. However, not presolving at all may hide features of a problem. For example, the size of the problem may be a great deal smaller in the case when variables turns out to be fixed after simple presolving, or the feature collector may state

that a problem is a MIP when in fact there is only one integer variable that has to be fixed after presolving, in which case it would be unreasonable to classify it as a MIP. Thus a balanced approach was taken, using only *trivial presolving* via SCIP 5.0. They define this trivial presolving as "removal of redundant constraints and fixed variables, activity-based bound tightening, and coefficient tightening".

As we will explain later, we decided to do very basic presolving routines when building our representation of the problem, which constitutes removing redundant inequalities and removing variables that could be substituted by a constant right away. The former may occur if, for example

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b_1 \quad \wedge \quad a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b_2,$$
$$\text{where } b_1 \leq b_2.$$

In such a case the second inequality does not add any information with its weaker bound, and we can remove it. The latter could happen if

$$x_i \leq b \wedge x_i \geq b,$$

which would result in fixing x_i to b in the solution, as well as substituting this fixed value in for x_i in all inequalities including it. Remark that this could, in special cases, lead to more redundant inequalities or fixed variables, and so it may be necessary to do presolving in several passes.

Infeasibility can also be decided right away if some bounds contradict each other, for example as in

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b_1 \quad \wedge \quad a_1x_1 + a_2x_2 + \cdots + a_nx_n \geq b_2,$$
$$\text{where } b_1 < b_2.$$

As all these types of bound checking turned out to be natural during the model representation step of our application, we decided to do this before recognition procedures are run. However, a parallelization of operations not needed in order to run recognition procedures could be experienced with. The following chapter will discuss more of the implementation, and design decisions, of the module created.

Chapter 3

Concept and Design

In this chapter we will describe how the module was created, and how the interface can be used. We will discuss pros and cons of different implementation decisions, and why we landed on the ones in the final implementation. The codebase can be found at [3], where the version we are discussing here has the commit hash ea6c9cd79799c369e6de4f12f6282ea5a1ee324f.

3.1 The Interface.

In the JuMP documentation it is strongly advised against implementing a solver interface for JuMP unless you really have to. If we could pretend our module was a solver, since it in theory would behave in the same way by either recognizing structures within the MIP before applying a connected algorithm, or by sending the problem to another solver, it would be ideal for it to seamlessly interact with JuMP via a solver interface.

However, as we realized this would be quite a task as the deeper workings of JuMP and MOI are not easy to get into without help from experts (it is recommended to join different forums if you decide to endeavor into this task). We decided that a more independent implementation would be enough in order

to discover if the concept will be a helpful tool, and if it will be suited for wrapping within a solver API at a later stage. Still, we will try to tie it strongly to the JuMP API, so that the usage of additional recognition algorithms would not provide much more work for a user already using JuMP for optimization.

The general method for optimizing linear programs with JuMP is to initiate a model with a solver optimizer object, either as a constructor argument, or attached to the model later with the `set_optimizer()` function. One can use specific macros to add variables, objective function, and constraints to the model. As we have seen, the general types of the constraints are function-in-set, where the function and set types supported are listed in **??**. For our use, we will generally want the `ScalarAffineFunction`, which is a real polynomial, in `LessThan`, which specify a constant the function should be less or equal to. Optionally one can set optimizer attributes through set-er functions. In the end one would call `optimize!()` with the JuMP model as argument, in order to start the solver on the problem.

For example, creating and solving the problem

$$\begin{aligned} &\text{Maximize} && x + 3y \\ &\text{Subject to} && \\ &&& 4x + 5y \leq 25.3 \\ &&& x \leq 3.2 \\ &\text{where} && x, y \geq 0 \quad \wedge \quad x, y \in \mathbb{Z} \end{aligned}$$

with the GLPK solver, with a time limit of 20 seconds, one would do the following.

Listing 3.1: Creating and solving a MIP.

```
using JuMP, GLPK
model = Model(GLPK.Optimizer)
@variable(model, 0 <= x <= 3.2, Int)
@variable(model, y >= 0, Int)
@objective(model, Max, x+3y)
@constraint(model, 4x+5y <= 25.3)
set_optimizer_attribute(model, "tm_lim", 20 * 1000)
optimize!(model)
```

Different solvers support different attributes, so how to use `set_optimizer()` is up to each solver to define, and need to be looked up in the solver's API documentation. This is one way JuMP, through Julia's multiple dispatch feature,

unifies API's while still providing flexibility for each solver. Thus, if one would make a solver API for our module at a later stage, choosing or setting recognition procedures would be intuitively done through `set_optimizer_attribute()`.

It is also possible to load problems from file, for example from a file in MPS format. The way to do so is to use MathOptInterface's submodule `FileFormats`. If we want to create a model with data from the file `file.mps`, we can import it as a special file model, and copy the data from this to an empty JuMP Model, with or without a solver attached. The following code in Listing 3.2 demonstrates it. Note that it is common practice to shorten the name of MathOptInterface to `MOI` within the code as well.

Listing 3.2: Reading a model from file.

```
using JuMP, MathOptInterface
const MOI = MathOptInterface
# Initiate empty MPS model:
file_model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MPS)
# Read data from file into MPS model:
MOI.read_from_file(file_model, "file.mps")
# Initiate an empty Model with GLPK solver attatched:
model = Model()
# Copy the data of the MPS model to the JuMP Model.
# Returns an index map, mapping corresponding variables of each model:
MOI.copy_to(model, file_model)
```

The flexibility of how one may create a model is something we want to keep at the user's end. Thus we need to figure out how we can accept and work with the model when our module receives it.

Figure 3.1 shows an overview of the components we landed on. Already existing is the `JuMP.Model` type, where the `Model()` constructor can be called by its own or with an optimizer factory, as we see in Listing 3.1 and Listing 3.2. An `Optimizer` is the interface to a solver, and it holds the pointers to the solver's inner representation of the model. A great amount of functions which take an `Optimizer` as an argument has to be implemented by the solver developers if they want to make the solver usable with JuMP. When a user calls different functions on a `Model` object, underlying functions can thus be called on the `Optimizer` in order to be able to set and read different attributes, solve the model, and get information about the results. In addition we have the `MathOptInterface` (MOI) layer, which works on more detailed and *unsafe* elements of the `Model`, which an average user not necessarily has to use. When a `Model` is created, a

`direct_model()` function is called, which works on MOI's backend type to store the data if an optimizer is not attached, or with an `Optimizer` if an optimizer is provided. As we saw in Listing 3.2, MOI also contains a lot of submodules which can be used for different tasks. `FileFormat` is one of them, another is the `Bridge` module which we will discuss in more detail, and there are others, for example one to help with benchmarking of wrappers.

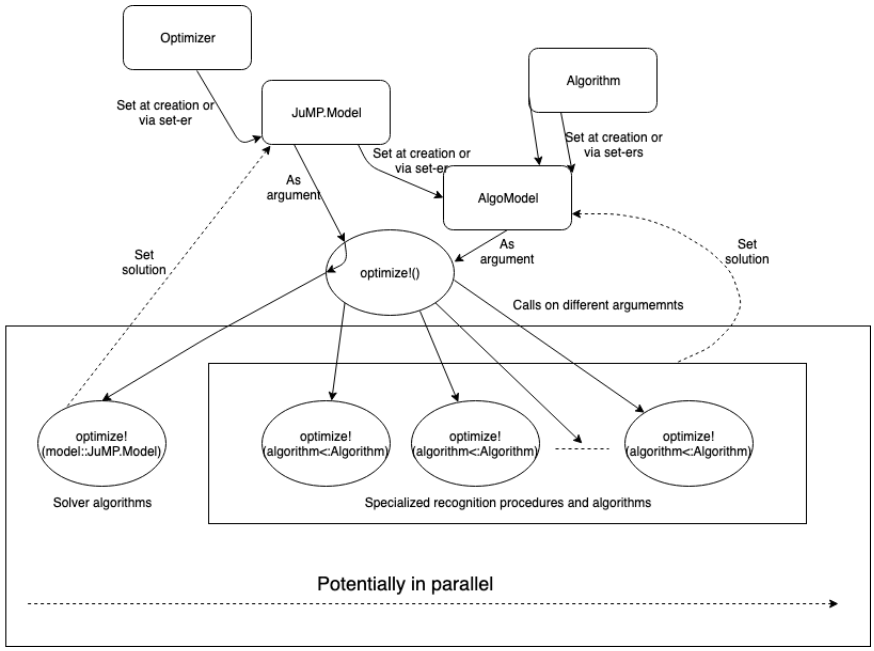

Figure 3.1: Concept diagram for MipFlex.

Since we are not wrapping JuMP, we work around this by creating the `AlgoModel` struct from a JuMP model. It will contain the JuMP `Model`, a representation of the model which is easier for us to work on directly, a vector of recognition procedures which should subtype a new type we named `Algorithm`, a solution struct, and a status enumeration type. The code definition of `AlgoModel` will look like the following.

Listing 3.3: `AlgoModel` struct.

```
mutable struct AlgoModel
    @atomic status::TerminationStatus
    jump_model::Union{JuMP.Model,Nothing}
    rep::Union{LPRep,Nothing}
    algorithms::Union{Vector{Algorithm},Nothing}
    @atomic solution::Solution
end
```

The constructor is made quite generic, so that one can initiate it without anything, with a JuMP model, with one or more algorithms, or with a combination of a JuMP model and one or more algorithms. For anything not set, the corresponding field will just contain `nothing`. The status will always be initiated as `Trm_NotCalled`, while the solution will be a new initiated `Solution` object, as defined below. The `status` and `solution` fields of `AlgoModel` have the `@atomic` macro in front, which indicates per-field atomicity when used in multithreading. We will explain briefly how this work in section 3.1.3.

The Solution struct is given as

```
mutable struct Solution
    @atomic primal_status::SolutionStatus
    @atomic x::Union{Vector{Float64},Nothing}
    @atomic objective_value::Union{Float64,Nothing}
    @atomic algorithm_used::Union{Algorithm,Nothing}
end
```

where the constructor initiates everything to `nothing` except the primal status, which will be set to `Sln_Unknown`. If a solution is found using one of the `Algorithm` objects, which we will explain how to create in section 3.1.2, then it is stored in the `Solution` struct. If a feasible or optimal solution is found, then it is set as the vector `x`. It is important to note that the indices in this vector is connected to the dictionary `var_to_name`, which is part of the `LPRep` object of `AlgoModel`, which we will in section 3.1.1. This dictionary has keys from 1 through the number of variables, and the entries are the names of the variables of the original model. If `"2 => t"` is an entry in this dictionary, it means that the variable with name `t` in the original model has a solution value in `x[2]` if a solution is set, where this `x` is the vector in the `AlgoModel` struct. The objective value should be set to a value whenever `x` is set and an objective function was given. Whether or not the objective function value is optimal, or just one of possibly many feasible solutions, can be discovered through reading the status of the `SolutionStatus`

enum. Finally, if a solution was found by one of the `Algorithm` objects, then the algorithm setting the solution should be reported in `algorithm_used`.

The `TerminationStatus` and `SolutionStatus` enums are very similar to what Tulip[4] uses, which is an LP solver written completely inside Julia, in contrast to most other solvers which are typically written in `C++`. Other solvers, however, use similar status, and we decided to incorporate the ones Tulip uses, with some extra added. It should be noted that `SolutionStatus` is to be used for both primal and dual solutions, but as it stands, we decided to only add a solution field for the primal solution. A possible extension is to add a solution field for the dual problem as well.

Listing 3.4: Status of `AlgoModel` and the primal solution.

```
"""
Termination status
"""
@enum(TerminationStatus,
    Trm_NotCalled,
    Trm_Unknown,
    # OK statuses
    Trm_Optimal,
    Trm_PrimalInfeasible,
    Trm_DualInfeasible,
    Trm_PrimalDualInfeasible,
    Trm_Feasibility,
    Trm_Infeasibility,
    # Limits
    Trm_IterationLimit,
    Trm_TimeLimit,
    # Errors
    Trm_MemoryLimit,
    Trm_NumericalProblem,
    # Others
    Trm_SolverUsed
)
```

```
"""
SolutionStatus
"""
@enum(SolutionStatus,
    Sln_Unknown,
    Sln_Optimal,
    Sln_FeasiblePoint,
    Sln_Infeasible,
    Sln_InfeasiblePoint,
    Sln_InfeasibilityCertificate,
    Sln_SolverUsed
)
```

For the `TerminationStatus` we can, for example, use `Trm_Unknown` in the case of a recognition procedure not recognizing a problem, say something about what kind of solution it is based on the dual, or report if the procedure reached some limit on the time or number of iterations. We added the statuses `Trm_Feasibility` and `Trm_Infeasibility` for use where either an objective function is not set, or when a recognition procedure is meant to be used only for looking for feasibility and does not take the objective function into account. These do not explain if it is the dual or primal problem which has a feasible solution, but for now we only work with the primal. The statuses ending in `SolverUsed` are set whenever a solver has solved the problem. In this case the user would need to use get-ers on the JuMP-model in order to find the solution. A solver would take over if

1. It has been connected to the JuMP-model, and
2. no algorithm of the AlgoModel object has found a solution if run sequential, or
3. the solver finds a solution faster in parallel.

An `Optimizer` can be added to the model after creation by either calling `set_optimizer(model::JuMP.Model,optimizer)`, or `set_optimizer(algoModel::MyModule.AlgoModel,optimizer)`. The thing is, since all references is made to the same JuMP-Model unless it is copied, calling either would update it. We added the function for the type `AlgoModel` simply for intuition, and in the case there would be any copying of models in a later release.

Whenever we have a `AlgoModel` created with a connected JuMP-model, and possible several `Algorithm` subtypes, we can call `optimize!()` on the `AlgoModel` object. If the program is initiated with 1 thread, it will apply the algorithms in the order they were added to the `AlgoModel`. If several threads are running, it will apply algorithms in parallel. As mentioned, whenever a solver is attached, it will be used to optimize the problem either in the end when sequential, or in parallel with the algorithms.

We will, in section 3.1.2, go through how an `Algorithm` subtype should be created, but in general, creating a new `Algorithm` subtype require the implementation of

```
optimize!(model::AlgoModel, algorithm::Subtype)
```

where `Subtype` is a subtype of `Algorithm`. `AlgoModel`'s `optimize!()`, i.e. the function where the signature only is defined by the single argument of an `AlgoModel` model, can thus use `optimize!()` on all the algorithms in turn, and on a solver if connected.

Whenever a solution is reached, appropriate statuses and solution fields should be set, ideally all other processes stop, and the function return. The parallel components are not fully created that way yet, so as of now, every thread run to completion. However, the multiple dispatch feature of Julia is used so that `optimize!()` is merely extended to be used on more types of models, where the arity and types of the arguments determine which function to be called.

An example of creating and solving a model is as follows.

Listing 3.5: Example use of our MipFlex module.

```
using JuMP, MipFlex, GLPK

model = Model(GLPK.Optimizer)
@variable(model, x <= 5, Int)
@variable(model, y <= 11, Int)
@variable(model, z == 3, Int)
@variable(model, a, Int)
@constraint(model, -y+4z <= 1)
@constraint(model, 3x+2y+z+2a <= 3)
@constraint(model, x-2y+5z+0.3a <= -4)
@constraint(model, 4x+9y-2z <= 0)

algorithm_vector = [.....]
algo_model = AlgoModel(model, algorithm_vector)
optimize!(algo_model)
```

where `algorithm_vector` is of type `Vector{{Algorithm}`, and is initiated to hold specific algorithms, also known as recognition procedures, which the user wants to try. We will show an example of such a recognition procedure in the next chapter.

3.1.1 Getting a Representation of the Model.

When not wrapping JuMP, it can be quite the challenge to read parameters of the model directly, as there is really no support from manually getting out the problem other than printing it, or using a combination of MOI get-ers and field access. For example, if one would want to look at the data of all constraints of the form `ScalarAffineFunction` in `LessThan`, that is, any polynomial required to be less than a constant, then one would need to do the following. First one can get a list of indices corresponding to the constraints matching this type of function-in-set

```
c_ids = MOI.get(lpmodel, MOI.ListOfConstraintIndices{
            MOI.ScalarAffineFunction{Float64},
            MOI.LessThan{Float64}
            }())
```

From these indices, one can get information about inequality i by calling specific get-ers on the backend of the model, which is a lower level type of a MOI model. If we want information about the terms of the affine function, we can call

```
con_fun = MOI.get(backend(model), MOI.ConstraintFunction(), c_ids[i])
```

From here we found no specific get-ers of the MathOptInterface implementation to get the actual data of the terms, and so we would need to use the struct's actual field names. There is a potentially major drawback of using the field names directly in our module. Since no function is provided, it was probably not in the developers mind that external users should access these fields, and so, if the field names get altered in a later release of MathOptInterface, we end up getting an error in the part of our code using it. However, as we are both aware of this, and at the same time make this module mainly as a workaround for the much bigger project of actually wrapping JuMP, we decided to take on this possible risk. As long as we know which version of JuMP it works with, we can always update our code as newer releases of JuMP or MathOptInterface are deployed.

So, continuing the example, a `ConstraintFunction` has terms, where each term has a `variable` and a `coefficient` field, where again `variable` has a `value` field. The `coefficient` is typically some real number representing the coefficient of the variable in this constraint, while the `value` refers to which index this specific variable is stored under. One can use the `var_to_name` dictionary of an `AbstractModel` type in order to get a mapping between variable index and the name registered within the model. Note that the naming strategy has nothing to do with what index a variable gets in the model. Also note that `var_to_name` is not even mentioned in the documentation of neither JuMP or MOI, and so we see again that we need to be aware of changes to the code since it was not intended for regular use. In order to get the variable index and the value of the coefficient of term j of constraint i, we can thus set

```
variable_index = con_fun.terms[j].variable.value
coefficient = con_fun.terms[j].coeficcient
```

where `con_fun` comes from the previous derivation above.

We see that it is not very straightforward to access the data of a model directly when not wrapping JuMP. This is one of the reasons why we decided to extract and save the data of the model in an own structure whenever an `AlgoModel` is created. In this way, any algorithm can use this ready data. For an even more flexible setup, one could let each algorithm-object provide its own function for extraction of data, if algorithm creators would want to customize this process further. However, at this stage we decided a pre-made model at an agreed upon form would be enough. This duplication of the model does come at a cost, both in processing time and memory usage. All the data has to be processed and stored both when a JuMP model is created, and also when the `AlgoModel` is created from this model. In addition, all the data is copied, so we double the memory usage. This could be a problem for very large LP's, so if used in larger scale, a wrapped module to use the same memory locations, as JuMP has intended for its solvers, would be beneficial. Altering the code to pass some data by reference may be preferable in cases of a lot of data as well.

When constructing a more accessible structure of the model data, we make use of the `MathOptInterface.Utilities.@model` macro, which makes it possible to define a custom `ModelLike` type, which implements the MOI model interface. By setting `is_optimizer=false`, it becomes a `GenericModel` type, which is a subtype of `AbstractModelLike`. As mentioned above, one can for example extract the `var_to_name` dictionary form this kind of type. The `@model` macro makes it possible for us to define which types of `Function-in-Set` constraints we support. By applying bridges, we can accept any data that can be transformed into the form specified. We wanted to do it as simple as possible, so we only allow `MOI.ScalarAffineFunction-in-MOI.LessThan`. For limiting the types of `VariableIndex-in-Set` constraints, that is, every constraint possibly added when a new variable is added to the model, we needed to implement `MOI.supports_constraints()` in order to forbid all the forms we did not want to accept.

So we created the new type `LPModel` with the `@model` macro, constricted the constraints to be `MOI.ScalarAffineFunction-in-MOI.LessThan`, and made a function for making a `LPModel` from a `JuMP.Model`. This function applies the `MOI.Bridges.full_bridge_optimizer` to an empty `LPModel` to define which bridges can be used for transforming constraints into the allowed forms specified by `LPModel`. Then it copies the JuMP-model into this `LPModel`. This assures that either the JuMP model is converted into the accepted constraints, or an exception will be raised. The `full_bridge_optimizer` contains most bridges except

for some specific non-linear ones, so the ones included are enough for our use.

When copying the data of a `JuMP.Model` into a `LPModel`, indirect calls to `MOI.supports_constraints()` are used in order to determine which variable constraints are supported, and then transform the variable constraints into accepted forms if possible. Variable constraints are stored in their own type of constraints, so a conversion for these has to be done separately. The version of `MOI.SupportsConstraints()` with LPModel as the input type will then be ran, namely the one we implemented in order to have control of which variable constraints to accept. The function defines which constraint types we do not directly accept, and we implemented it as follows

Listing 3.6: Variable constraints not supported.

```
function MOI.supports_constraint(
    ::LPModel{T},
    ::Type{MOI.VariableIndex},
    ::Type{<:Union{MOI.Interval{T},MOI.Semicontinuous{T},
    MOI.Semiinteger{T},MOI.ZeroOne}}) where T
    return false
end
```

The variable constraint sets not listed as types in this function will thus be supported by `LPModel`. They are `MOI.GreaterThan{T}`, `MOI.LessThan{T}`, `MOI.EqualTo{T}`, and `MOI.Integer`.

For example, we do not accept a variable to be stored as an own interval type, and so such a constraint will be converted into a `MOI.LessThan` and a `MOI.GreaterThan`. That is, we get the transformation

$$x \in [a, b] \quad \rightarrow \quad x \geq a \wedge x \leq b,$$

where $a, b \in \mathbb{R}$, and x is a variable of some LP. As a clarification, `MOI.LessThan` and `MOI.GreaterThan` are defined as containing the connected constant, so that it is actually less-or-equal and greater-or-equal.

Similarly, `MOI.ZeroOne` can be converted to our model by requiring the variable to be integer, less or equal to 1, and greater or equal to 0. A semicontinous or semiinteger set cannot, in most cases, be supported by our model, as these are defined as an interval of reals or integers, respectively, together with the set containing only zero. Whenever zero is not contained in the interval, this union cannot be represented with our definition, and so a `MOI.UnsupportedConstraint`

exception will be thrown. This is something we easily can adjust in a later iteration, but for this prototype we keep it fairly simple.

At this stage we know the possible forms of all the constraints, and so it is much easier to extract the data. For making the further job of constructing `Algorithm` subtypes a bit easier, we defined a struct `LPRep` to hold the necessary data for an algorithm. When constructing `LPRep` some simple preprocessing and cleaning up is also done.

The LPRep has quite a lot of fields, so we will start by describing each shortly.

- `is_consistent::Bool`
  - A boolean set to false whenever simple inconsistencies are found.
  - Such an inconsistency can be that a single variable is required to be both less than a constant, a, greater than a constant b, and $a < b$.
- `var_count::Int64`
  - The number of variables in the model.
- `con_count::Int64`
  - The number of polynomial constraints in the model. This count does not include constraints on single variables.
  - Note that the original model may have had more constraints originally, but after preprocessing, this number is altered to reflect the numbers of rows in `A` (see below) of `LPRep`.
- `sense::MOI.OptimizationSense`
  - What kind of optimization we are looking at. Are we looking for a feasible solution, or is there an objective function to maximize or minimize?
  - Default is `MOI.FEASIBILITY_SENSE`, but `MOI.MAX_SENSE` or `MOI.MIN_SENSE` is set whenever an objective function is given.
- `c::Vector{Float64}`
  - Coefficient of each variable in the objective function.

  - Variable with index i will have its corresponding coefficient at index i within this vector.

- `obj_constant::Float64`
  - The constant term of the objective function, if there is any.

- `A::SparseMatrixCSC{Float64, Int64}`
  - A sparse matrix representing the left-hand-side of the constraints of the LP.
  - It is column major, and so getting a column of values is the most efficient.

- `b::Vector{Float64}`
  - A vector representing the right-hand-side of the constraints. It can be thought of as a column vector.
  - Index j of `b` represents the right-hand-side of the constraint with left-hand-side consisting of row j of matrix `A`.

- `At::SparseMatrixCSC{Float64, Int64}`
  - A sparse matrix equal to the transpose of `A` from above.
  - Whenever we want to examine rows of `A`, it is faster and easier to look up corresponding columns of `At`. This convenience is sacrificed for memory usage.

- `greater_than::Dict{Int64, Float64}`
  - Dictionary containing variable indices as keys, and lower bounds as values.
  - If `greater_than[i] = j`, it means that variable with index i is required to be greater or equal to the value `j`.

- `less_than::Dict{Int64, Float64}`
  - A dictionary similar to `greater_than`, but now the values are upper bounds instead.

- `integer::Dict{Int64, Bool}`

  - A dictionary where the keys indicates which variables has to be integer.
  - A dictionary is used mainly because it then is consistent with the other variable constraint data types in `LPRep`, and it is a dynamic data structure, so that the preprocessing can be done without knowing how many elements it is required to hold.

- `equal_to::Dict{Int64, Float64}`
  - Dictionary containing variable indices as keys, and values they have to be equal to as values.
  - During preprocessing, combining `less_than` and `greater_than` can for example result in new variables having to be equal to a value. Thus this flexible data structure is handy.

- `var_to_name::Dict{Int64, String}`
  - The aforementioned dictionary holding the variable indices as keys, and the names given when the JuMP model was created.

Any recognition procedure working on a LP should be able to get the desired data from this. Our current implementation assumes direct access to all fields, except when setting the solution and termination status, which should be done through dedicated functions to assure atomicity, and avoid race conditions when multithreading.

It is probably most inconvenient to retrieve data from the sparse matrices. However, as each constraint often have few variables involved, a sparse matrix representation will save a lot of space and time when working with big problems, as there often are a lot more zero entries than non-zero. The Compressed Sparse Column (CSC) Sparse Matrix Storage of the JuMP module `SparseArrays` is used, and there are some useful functions one can employ in order to get and alter the data of a sparse matrix. A vector of all the nonzero elements of `A` can be returned when calling `nonzeros(A)`, where `A` is a `SparseMatrixCSC`. A vector of corresponding row indices can be retrieved with `rowvals(A)`. This means that `rowvals(A)[i]` gives the index of which row `nonzeros(A)[i]` lays in, where `i` is an integer in the range of nonzero values of `A`. The function `nzrange(A, j)` gives the range of all indices that points to values belonging to column `j` of `A`. Thus, if `i` $\in$ `nzrange(A, j)`, then the value `nonzeros(A)[i]` is located at

coordinate `(rowvals(A)[i], j)` within `A`, where the first coordinate indicates row number, and the second column number. From this structure it follows that going through each column in a matrix is easy. A an example from JuMP, we can do the following.

```
A = sparse(I,J,V)
rows = rowvals(A)
vals = nonzeros(A)
m, n = size(A)
for j = 1:n
   for i in nzrange(A, j)
      row = rows[i]
      val = vals[i]     # value of (row, j)
      # perform sparse wizardry...
   end
end
```

Going through rows would be much harder, as we would need to do something as

```
A = sparse(I,J,V)
rows = rowvals(A)
vals = nonzeros(A)
m, n = size(A)
for i = 1:m
    row_i = findall(v -> v==i, rows)
    for index in row_i
        for j = 1:n
            if index in nzrange(A, j)
                val = vals[index]   # value in (i,j)
                 # perform sparse wizardry...
            end
        end
   end
end
```

Here we need to call `nzrange(A, j)` `n` times per element instead of only `n` times, as well as search up all indexes of elements residing in row `i` before starting. Thus we need `m` times more calls to `nzrange(A, j)`. In addition, it is less intuitive for a programmer to apply the latter approach, and with these arguments we will justify saving the transpose of `A` in `LPRep`. If we need to search through rows of `A`, we simply search through columns of `At`. It is worth mentioning that making `At` directly from `A` is not that simple, but it is very simple when constructing the matrices. We only need to switch the first two vector arguments

of the `sparse(R,C,V)` function, where `R` and `C` represent row and column indices respectively, and `V` indicates the values.

As mentioned under the `num_constraints` field, this number can differ from the initial input of constraints. We also have the `is_consistent` field, where this field set to **`false`** indicates a detected inconsistency. These results comes from the simple preprocessing of the initial problem.

The preprocessing consists of first applying a simple consistency check on the `SingleVariable` restrictions. Secondly, if any variable with index `i` has to be bound to a specific value, we substitute this out of the constraint matrix, `A`, and update this matrix to not include column `i`, but rather move these constant values of this column to the right hand side, changing the vector `b` accordingly. Whenever such a pass is done, any row of constraints can end up with only one variable left, and if this is the case, we update the `less_than` or `greater_than` dictionary if this bound is to be stricter than the previous one. We also remove the rows of only 1 variable, as the dictionaries with single variables will then hold this information. At this step we should do a new consistency check on the variables. There is also a possibility that a stricter bound results in a `less_than` and a `greater_than` value for the same variable index to become equal, meaning we have a new variable that has to be equal to a certain value. Thus we need another pass of substituting out these potentially new bound variables. We continue to do such passes until no more constraints contain only one variable, or no constraints are left. However, whenever a consistency check has detected a flaw, it will set the `is_consistent` flag to **`false`**, and stop building `LPRep`, setting all the fields except `is_consistent` to `nothing`.

The consistency check performs four simple checks. The first three tests are done for every variable whose index is contained in both the `less_than` and the `greater_than` dictionary. The first test simply checks whether

```
less_than[i] < greater_than[i],
```

which will obviously make the set of possible values for this variable to be empty.

Then we have some tests for whenever the two bounds (upper and lower) are equal,

```
less_than[i] == greater_than[i],
```

in which case the variable has to be equal to this new value. Let `c` be this value. If a different value is already set in `equal_to[i]`, we reach a contradiction, and

thus `is_consistent` is set to **`false`**. If `integer[i]` is set, then we also need to check if `c` is integer, and assign **`false`** to `is_consistent` if not. Finally we update the variable dictionaries as follows

```
equal_to[i] = less_than[i]
delete!(greater_than, i)
delete!(less_than, i)
```

since we now rather has an equality constraint.

The third case we inspect is whenever

$$\texttt{greater\_than[i]} < \texttt{less\_than[i]}.$$

The only thing we need to check here, is if `integer[i]` is set, and then the interval `[greater_than[i], less_than[i]]` has to contain at least one integer. The way we check this is to see if neither bound is integer, and if rounding down each bound to nearest integer provide the same number. In this case both bounds are between two integers, and so the interval defined by them cannot contain any integers, in which case `is_consistent` is set to **`false`**.

The final check is to see whether a variable that is required to be equal to a value, also has either a lower or upper bound. If so, we check if either

$$\texttt{less\_than[i]} < \texttt{equal\_to[i]},$$

or

$$\texttt{equal\_to[i]} < \texttt{greater\_than[i]},$$

which will lead to contradictions.

We see that this kind of presolving is very basic, none concerning constraints with more than one variable. Since our goal is more concerned with recognition procedures, we will rather let the presolving be simple, since it always will run before any recognition procedures can start, and thus should not take too long time. We will rather focus on how we could add recognition procedures. Such procedures could for example run other types of presolving methods. However, if such procedures where to alter the representation of the problem in `AlgoModel` directly for other recognition procedures to use, then some may break since they in general would expect the form we have described. If the procedures are run in parallel, altering the `LPRep` directly without any form of communication can

also undermine other procedures. So at this stage, any recognition procedure should only work on a local copy of the data.

One potential weakness of the current design is that for any small change to the model, one would need to build the entire `LPRep` from scratch. This can be done using the `update!(a::AlgoModel)` function if the inner JuMP model should be changed. The documentation of how to make a solver wrapper addresses this matter for optimizes as well, and prompts developers to choose if they should have support for incremental modification of the model or not. Incremental modification allows for the user to add single variables or constraints without needing to rebuild the entire problem, and alter data after an `optimize!()` call. Obviously the solver has to have support for this in order to make an interface for it, and not all solvers support this. JuMP also go on recommending to not support this feature if it is the first time the developer has implemented an interface for a solver. From this we gathered that it may not be the simplest task to support incremental changes, so we let this be a possible upgrade in the future.

3.1.2 Adding Recognition Procedures

Now we have a skeleton for our code, and an agreed upon representation of the problem. Thus we only need to document how a user can add her recognition and solving procedure. There are some requirements so that the interface can work, but we also have some suggestions for how to structure the code in order to make the process more intuitive.

Let us make an example where one would like to implement a recognition procedure which we call `MyAlgorithm`. First we have to declare a struct which will be a subtype of `Algorithm`. The subtyping is mostly for readability, but also to make sure that nothing else than `Algorithm` objects can be put into the `AlgoModel`. You have to decide whether or not the procedure needs additional information. If so, add the needed fields to the struct. Say that we want to be able to pass a time limit on the total time this algorithm uses. Then we can for example do the following

Listing 3.7: Creating an algorithm.

```
struct MyAlgorithm <: Algorithm
    time_limit::Union{UInt128, Nothing}
end
MyAlgorithm() = MyAlgorithm(nothing)
```

This will make sure that if no limit is passed, it is saved as `nothing`. In the implementation of the algorithm we can check for the limit, and if it is not set, we can either use a default limit, or no limit, depending on what behavior we would want.

Secondly, the bare minimum is to implement the function

Listing 3.8: Implementing optimize function.

```
optimize!(algo_model::AlgoModel, algorithm::MyAlgorithm)
```

This procedure is meant to look through the LP of `algo_model`, see if the data matches a specific pattern, and run a tailored algorithm for this case. If the algorithm does not recognize the problem, and thus cannot set any solution, it should set the termination status to `Trm_Unknown`, and the solution status to `Sln_Unknown` through the following functions.

Listing 3.9: Setting status.

```
set_trm_status!(model, Trm_Unknown)
set_sln_status!(model, Sln_Unknown)
```

These set-ers are made to make sure that race conditions do not occur if the program is run in parallel and several procedures find an answer at the same time.

If an answer is found, corresponding termination status and solution should be set. Whenever an an algorithm is only concerned with feasibility vs infeasibility, then use the corresponding termination statuses. If both the primal and the dual has one optimal solution, `Trm_Optimal` is used, whereas when one is bounded and the other is infeasible, the `Trm_PrimalInfeasible` or `Trm_DualInfeasible` are used, indicated which is infeasible. If both are infeasible, use `Trm_PrimalDualInfeasible`.

A solution can be set with

Listing 3.10: Setting solution.

```
set_solution!(model::AlgoModel,
              primal_status::SolutionStatus,
              x::Union{Vector{Float64}, Nothing},
              objective_value::Union{Float64, Nothing},
              algorithm_used::Union{Algorithm, Nothing})
```

where the order of variables in `x` should be the same as they have indices in the columns of `A` in `lprep` of the `AlgoModel`. The objective value will then be

```
dot(lprep.c, x) + lprep.obj_constant
```

where `dot` is the scalar product operation. Do also log the algorithm object that `optimize!()` was called with when the solution is set. That way the user will know which of the procedures recognized the problem and found the solution. Make sure to return **`true`** whenever a solution is set, and **`false`** otherwise. The main `optimize!()` on an `AlgoModel` object will continue calling `optimize!()` on algorithm objects until it receives **`true`**.

When implementing the `optimize!()` function, we recommend gathering everything that has to do with recognizing a specific problem into a recognize function, taking in the model and your algorithm object as arguments. That way the you can start by a simple call to recognize in order to see if the problem structure you are looking for is found. If the problem is not recognized as a matching structure, return **`false`** immediately. If it recognized, continue on with the intended algorithm in order to find a solution.

Note that it is not implemented any get-ers for the `lprep` object at this stage. For now we let the recognition procedure retrieve all the fields directly. For a larger deployment one would want to have get-ers, in the case where one would alter the inner code, so that user's code not will break. However, as this is a prototype, we let that task be postponed to a later stage.

3.1.3 Parallel Components

Being able to let an `AlgoModel` object run `optimize!()` in parallel on all the `Algorithm` objects connected to it, and on the JuMP model if a solver is attached, would be very beneficial, as the thread first finding a solution could halt the other processes. In that way, a procedure which could find the solution relatively fast, does not have to, in the worst case, run after all the other recognition procedures.

At this stage of the development, we have only included a check, in the `optimize!(::AlgoModel)` function, of how many execution threads are currently active in Julia. If there are more than 1 thread, then the code looping through different versions of the *optimize*!() functions, on the algorithms and on the JuMP model, will be run with macros from the Julia Threads package, making sure that the tasks are divided among threads. The fields containing the solution and status are initiated and set via the `@atom` macro, making sure the altering of fields are done atomically. In addition, the termination status is checked before setting it, so that threads do not try to do this at the same time. However, we have not currently tested and made sure that there are no potential concurrency issues, so per now, this feature is mostly a prototype. The `@threads` macro provide no easy way of killing off threads from other threads either, which would be crucial in a parallel application. The whole point is to reduce running time, and stop when the first procedure finds an answer to the LP.

3.1.4 Notes on Testing

When developing the code, we have made sure to constantly write unit tests during implementation, trying to cover all functionality. However, when running through MIPLIB's benchmark set, there did arise some problems not being accounted for in the unit tests. Thus testing on a bigger set of real world problems was a good way of complement the unit tests.

One issue that came forth when testing on MIPLIB, was that for a certain problem (specifically `bab2.mps`), `AlgoModel` returned `Trm_Infeasible`, even though the problem was hard but feasible. It turned out that it was deemed infeasible during the preprocessing stage when doing the passes of substituting bound variables in the constraint matrix, and rewriting the constraints. The issue here was that when checking whether or not

$$\texttt{less\_than[i]} < \texttt{greater\_than[i]},$$

for a variable with index `i`, then, even though these values were supposed to be equal in theory, lack of precision, due to floating point arithmetic during the rewriting of constraints and bounds, lead to evaluating the statement above to be true. This in turn sets the `is_consistent` flag to **`false`**, which will lead to a registration of an infeasible solution.

This behavior, of course, breaks the reliability and logic of the code drastically. How we temporally have fixed it is checking whether or not

$$\texttt{less\_than[i]} \approx \texttt{greater\_than[i]}$$

first, and if so, deem the bounds equal. If, on the other hand, two comparable bounds should happen to not be theoretically equal, but still within the range of the approximation binary operation to be deemed equal, then we will have a problem with this workaround. Further work on this MipFlex should address this matter, and see if there should be rewritten, or included, some kind of error estimate in the returned solution.

Chapter 4

Recognition Procedures

We reduced the scope of this project to looking into details of one type of recognition procedure. Discussion of more possibilities are discussed in the end of this text.

4.1 System of Difference Constraints

4.1.1 The Basic Case

A difference constraint is a constraint on the form

$$x - y \leq t, \tag{4.1}$$

where x and y are variables, while t is some constant. We will start by looking at the general case where variables and constants are in $\mathbb{R}$. Constraints like the one in eq. (4.1) can for example occur when x and y represent the time of some events e_x and e_y respectively, and the constraint thus states that e_x should occur within time t of event e_y.

Equivalently, the expression

$$-x + y \leq -t, \tag{4.2}$$
$$\Leftrightarrow x \geq y + t \tag{4.3}$$

could express that event e_x should occur after time t of the start of event e_y.

Look at a system of m difference constraints over n different variables, that is, where all constraints are of the form

$$x_j - x_i \leq b_k, \quad \text{where } i, j \in [1, n], \text{ and } k \in [1, m]. \tag{4.4}$$

In matrix form we would get

$$\mathbf{Ax} \leq \mathbf{b},$$

where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, and $\mathbf{A}$ is a $m \times n$ matrix where row k would have one 1 at index j, one -1 at index i, and the rest of the entries being 0. Recognizing a system of normalized difference constraints simply becomes checking if each row of $\mathbf{A}$ contains exactly one 1 and one -1, while the other entries are zero. If we have a sparse matrix representation, and easily can check each non-zero values of every row, then this will take $\mathcal{O}(m)$ time.

For such a linear program of difference constraints one can define the related *constraint graph* as follows.

Definition 1. *Let $\mathbf{Ax} \leq \mathbf{b}$ be a system of difference constraints, with constraints on the form Equation 4.4. The **constraint graph**, $G = (V, E)$, is given as*

$V = \{v_0, v_1, \ldots, v_n\}$,
$E = \{(v_i, v_j) | x_j - x_i \leq b_k \text{ is a constraint}\} \cup \{(v_0, v_l) \mid v_l \in V \setminus \{v_0\}\}$,
with $i, j, l \in [1, n], k \in [1, m]$,
where the weight of each directed edge is given by
$w(v_i, v_j) = b_k$, when $x_j - x_i \leq b_k$ is a constraint, and
$w(v_0, v_i) = 0$, otherwise.

If the graph has no negative cycles, finding the shortest path from v_0 to each vertex will provide a solution to the linear program. In fact, the program will be feasible if and only if there exist shortest paths in the difference constraint graph.

Proposition 4.1.1. *Let $G = (V, E)$ be a constraint graph formed from an LP of difference constraints, as in Definition 1. The LP is feasible if and only if there exists a shortest path to every vertex v_i, $i \in [1, n]$, from v_0. Moreover, the vector*

$$\mathbf{x}^T = [\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n)]$$

is a solution to the LP, where $\delta(v_0, v_i)$ denotes the shortest distance from v_0 to v_i.

Proof. We will start by showing that if there exists a solution to the shortest path problem, then

$$\mathbf{x}^T = [\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n)]$$

is a solution to the LP, and thus it is feasible.

Since a shortest path to v_j is no greater than any other path to v_j, specifically a shortest path to any other vertex v_i plus the edge from v_i to v_j, the triangle inequality is a consequence. We have that

$$\begin{aligned} & \delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j) \\ \Leftrightarrow \quad & \delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j) \\ \Rightarrow \quad & x_j - x_i \leq b_k \quad \text{is satisfied if } x_t = \delta(v_0, v_t) \quad \forall t \in [1, n]. \end{aligned}$$

Thus every LP constraint, which is bijectively mapped with shortest distances in the graph, will be satisfied.

Now, for the other direction, assume, by sake of reaching a contradiction, that the LP is feasible while there does not exist a solution to the shortest path problem. Since there exists an edge (v_0, v_i), with $w(v_0, v_i) = 0$, for any edge $v_i \in V \setminus \{v_i\}$, the shortest distance to any edge is at most 0. Thus, no solution to the problem means that at least one edge is unbounded from below. This will happen only if there exists a negative cycle in the graph, so that it can drive the weight down infinitely.

Assume without loss of generality that a negative cycle is given by

$$v_1 \to v_2 \to \ldots \to v_p \to v_1.$$

By definition of a negative cycle the sum of all the edges are strictly less than zero

$$\left(\sum_{i=1}^{p-1} d(v_i, v_{i+1})\right) + d(v_p, v_1) < 0$$
$$\Leftrightarrow \quad \sum_{k=1}^{p} b_k < 0,$$

where we use the corresponding b's of each edge. However, if we express the sum of these $b's$ in relation to the inequalities of the linear program, we get that

$$(x_2 - x_1) + (x_3 - x_2) + \cdots + (x_p - x_{p-1}) + (x_1 - x_p) \leq b_1 + b_2 + \cdots + b_p$$
$$\Leftrightarrow \quad 0 \leq \sum_{k=1}^{p} b_k,$$

which is not possible to obtain if there is a negative cycle. As such we have reached a contradiction. □

Proposition 4.1.1 suggests the use of Bellman-Ford for solving the system, as it finds the shortest distance from the source, v_0, to each vertex, as well as determining if there exists any negative cycle reachable from the source vertex. In this case it will detect any negative cycle, as all vertices are reachable from v_0 by construction. The complexity of Bellman-Ford is $\mathcal{O}(VE) = \mathcal{O}((n+1)(m+n))$, since we added a vertex v_0 and edges from it to all the n other vertices. However, these components were artificially imposed in order to simplify the arguments, but we can observe that they are not needed in the actual algorithm. We know that after a first relaxation, all vertices would gain the distance 0 from their edge from v_0. We also know that there are no incoming edges to v_0, so no other path via v_0 would improve the distances. This implies that we can initiate all vertices to have distance 0, and remove v_0 and all its incident edges from further search. One ends up running the algorithm over n vertices with m edges instead, and the run time for solving the LP will be $\mathcal{O}(nm)$.

Forms of Inequalities.

In theory, one could add as many inequalities one would like, and make an edge for every single one of them. However, it is only necessary with the strictest bound for the same linear combination of variables. That is, for x_i and x_j having several upper bounds on $x_j - x_i$, we only need the lowest upper bound in order to cover all of the restrictions.

$$\begin{aligned} x_j - x_i &\leq b_1 \\ x_j - x_i &\leq b_2 \\ &\vdots \\ x_j - x_i &\leq b_c \\ &\Updownarrow \\ x_j - x_i &\leq \min_{k \in [1,c]} b_k. \end{aligned}$$

The only other allowed linear combination for x_i and x_j when normalized, is then $x_i - x_j$. Since we demand inequalities as input, this can be needed to express equality, since

$$\begin{aligned} x_j - x_i \leq b_k \quad &\wedge \quad x_i - x_j \leq b_k \\ &\Updownarrow \\ x_j - x_i \leq b_k \quad &\wedge \quad -(x_j - x_i) \leq b_k \\ &\Updownarrow \\ x_j - x_i &= b_k, \end{aligned}$$

or it can be a requirement creating a legal interval which $(x_i - x_j)$ can reside in:

$$\begin{aligned} x_j - x_i \leq b_k \quad &\wedge \quad x_i - x_j \leq b_l \\ &\Updownarrow \\ x_j - x_i \leq b_k \quad &\wedge \quad x_j - x_i \geq -b_l \\ &\Updownarrow \\ x_j - x_i &\in [-b_l, b_k]. \end{aligned}$$

We can also observe that if $b_k < -b_l$, then the problem is automatically infeasible, as there are no legal values for the difference $(x_j - x_i)$.

In any case, for several inequalities of the type $(x_i - x_j)$, we can do the same as for $(x_j - x_i)$, namely limit the amount of them to one, for the lowest upper bound. Thus, for any two variables, we can have at most two inequalities, resulting in two edges in the constraint graph, pointing in opposite directions. The maximum number of inequalities is then two times the number of variable combinations

$$2\binom{n}{2} = 2\frac{n!}{2!(n-2)!} = \frac{n!}{(n-2)!} = n \cdot (n-1) = \mathcal{O}(n^2),$$

and so the upper bound on Bellman-Ford runtime becomes $\mathcal{O}(n^3)$.

Property of the Solution

It is typically not connected any specific objective function for maximization or minimization over systems of difference constraints; one rather looks for feasibility. However, if one looks at scheduling jobs, it would be nice to know what the shortest amount of time one would need in order to finish all jobs. For example, if the algorithm gives a feasible solution, it still might not be preferable to get a solution where the different jobs are scheduled thousands of years apart when they could be done in two hours. Luckily, since the Bellman-Ford uses edge-lengths equaling the constraint bounds, which is typically mapped from the time needed between job starts, it finds solutions as close to the bounds as possible. More precisely, we are maximizing $min\{x_i\}$, meaning that we are using the maximum edge lengths possible without the difference constraints being violated, but minimizing the distances over the graph in order for all inter-dependent variables to fulfill all constraints. We could have gotten a legal solution by setting edges less than the b_k's, since any lower value is also possible, but we set them equal to the b_k's, so that the edges between vertices will be the larges we are allowed to have. Thus we are forcing out the maximum value of the distances, while still having them as small as needed in order to not violate the constraints. Observe further that $max\{x_i\} = 0$. That is, the maximum distance is 0 due to the initialization of distances and the fact that updates can only strictly decrease the distances. In addition, at least one vertex has to have distance 0, since if no vertex had a shortest distance of 0, then all vertices would have a predecessor that were not v_0. Reconstructing the path from any vertex by using predecessors would never reach v_0, contradicting the fact that we are finding shortest distances from v_0 with Bellman-Ford. Thus

$max\{x_i\} = 0$. Combining these observations, we get that

$$\begin{aligned} & max(min\{x_i\}) \\ & = min(-min\{x_i\}) \\ & = min(0 - min\{x_i\}) \\ & = min(max\{x_i\} - min\{x_i\}), \end{aligned}$$

and so, if there exists a feasible solution, the solution given by Bellman-Ford minimizes the distance between the highest and lowest variable value. If these values represent starting times of events, it means that the time between starting the first and last task is minimized, spending the least amount of time, which would be preferred.

4.1.2 Imposing Integrality Constraints

Now let us specify the problem further. What can one say about a system of difference constraints when one restricts all or some of the variables in the system to be integers? The closest we find regarding other work on this is an exercise in Cormen[10](exercise 24.4-12) which asks for a solution to this exact problem. As this is an exercise meant for students of the algorithmic course, it may suggest that the solution should be quite straightforward to find and prove, even though it is marked as a more demanding exercise. However, as we started working with it, it did not seem quite as straight forward as one may assume at first glance. Upon consulting the accompanying instructor's Manual of Magnus Hetland, we discovered that the solution is omitted from it. There exists a GitHub page dedicated to collecting answers to the exercises of the book, but here is only a short suggestion to the solution made, and no proof backing it up. As these contributors are made up of others than the writer of the book, and there are no solid proofs, we see it helpful to contribute with a more solid argument for a solution to the problem. And as we will see, the solution suggested on the GitHub page, regarding rounding down path lengths whenever encountering a node corresponding to an integer variable, is not fully complete either, as the behavior of negative cycles when rounding is not as expected. It turns out that we have no guarantee that the problem is infeasible if it is not found a solution after $n - 1$ iterations, as it is with Bellman-Ford, where n is the number of nodes in the graph.

Let us look at the problem. It is apparent that the path length depends on the distances between vertices, $d(v_i, v_j)$. If all the distances are integral, that is, the vector $\mathbf{b}$ of the linear system $\mathbf{Ax} \leq \mathbf{b}$ has all entries integral, all distances would be integral as well. One can see this by the fact that the Bellman-Ford algorithm operates by propagating distances, and all integral distances would result in all shortest paths being integral from this propagation.

When a b_k is fractional, one has four possibilities regarding the corresponding variables, x_i and x_j, in the inequality. Let us look at all of these cases:

1. **Both variables have to be integers.** In this case $x_j - x_i$ has to be integral, and so rounding the upper bound, b_k, down to the nearest integer, $\lfloor b_k \rfloor$, will not remove any legal solution.

2. **Both variables can be fractional.** Here there is no issue with the upper bound being fractional, so we can let the inequality be as it is.

3. **Only the "tail", x_i, has to be integral.** Since the corresponding graph element for the inequality $x_j - x_i \leq b_k$ results in a directed edge from v_i to v_j of distance b_k, it does not matter if a fractional shortest path propagates *after* an integral path has been reached to the vertex v_i.

4. **Only the "head", x_j, has to be integral.** Here there is a potential problem, as there could be a fractional shortest path to v_i, which could propagate through (v_i, v_j) and remain fractional at v_j. Could one still round down the value at v_j, without cutting away any solution for v_j or other vertices with distances propagating from v_j? We will argue that the answer to this is 'yes'. However, we can no longer guarantee that the algorithm will conclude with infeasibility correctly after $n-1$ iterations, and we will show an example of why it is so.

First, let us look at the regular Bellman-Ford, as it is presented in Cormen. Let $G = (V, E)$ be a graph, with a connected weight function $w : E \to \mathbb{R}$, and a source vertex s. Let V and E be attributes of G, giving the vertex and edge set of G respectively. Let d and p be attributes of each vertex, where d is maintaining the upper bound of a shortest path found to the vertex from the source, and where p is the predecessor vertex in such a path. The original Bellman-Ford algorithm follows the following pattern in order to find shortest path to all vertexes from the source.

Algorithm 2 Bellman-Ford

```
BELLMAN-FORD(G, w, s)
 1: for each vertex v ∈ G.V
 2:     v.d = ∞
 3:     v.p = NIL
 4: s.d = 0
 5: for i = 1 to i = |G.V| − 1
 6:     for each edge (u, v) ∈ G.E
 7:         if v.d > u.d + w(u, v)
 8:             v.d := u.d + w(u, v)
 9:             v.p := u
10: for each (u, v) ∈ G.E
11:     if v.d > u.d + w(u, v)
12:         return FALSE
13: return TRUE
```

It is proved, for example in Cormen, that if the algorithm returns 'true', then there are no negative cycle reachable from the source, and for all vertices $v \in V$, $v.d$ is the length of the shortest path from s to v. Furthermore, by recursively following the parent attribute, p, from v until reaching s, it will give a path of length $v.d$, where the length is given by summing up the weights of the edges in the path. It is also proven that if the algorithm returns 'false' then there is a negative cycle reachable from the source, and there is no finite solution to the problem.

The convergence property is used to show that the Bellman-Ford algorithm will find all reachable shortest paths that exists after $|G.V| - 1$ iterations. Because of this guarantee, one can instead implement the algorithm as

Algorithm 3 Bellman-Ford, no final relax

```
BELLMAN-FORD(G, w, s)
 1: for each vertex v ∈ G.V
 2:     v.d = ∞
 3:     v.p = NIL
 4: s.d = 0
 5: changed = FALSE
 6: iterations = 0
 7: do
 8:     changed = FALSE
 9:     for each edge (u, v) ∈ G.E
10:         if v.d > u.d + w(u, v)
11:             v.d := u.d + w(u, v)
12:             v.p := u
13:             changed = TRUE
14:     iterations += 1
15: while changed and iterations < n
16: if changed
17:     return FALSE
18: return TRUE
```

where one stops either after no more relax operations are needed, and thus a result is reached, or the maximum number of iterations is done. If the maximum number of operations is done, but one could still relax distances, then there has to be a negative cycle, and no feasible finite solution exists.

Adjusted Bellman-Ford

Let us first assume that we can run infinitely many iterations, and argue that if there is a solution to the integer problem, it will be found when we do the suggested solution of rounding whenever relaxing along an edge with an integer node as its head. We will do the following alterations to algorithm 3.

Initialization. First off, as briefly touched upon, due to construction of the constraint graph, all vertices will have a shortest path with a value of less or equal to 0. This is because all vertices have a path from v_0 of length 0, and can only decrease in value in the relax operation of the algorithm. Thus it is rational to start out with the d attribute set to 0 as an upper bound. It should also be accepted to remove all edges from v_0 from further search, as there are no edges going back to v_0 that can improve the distance via it. That is, the condition for updating any vertex distance v_i via v_0,

$$v_i.d > v_0.d + w(v_0, v_i),$$

will never happen. v_0 is always 0, since there are no edges into v_0 that can update $v_0.d$, $w(v_0, v_i) = 0$ from construction, and $v_i.d = 0$ by initiation, and can only decrease, since it can only be updated via the relax operation into strictly smaller values. Using these observations, we will always have

$$v_i.d \leq v_0.d + w(v_0, v_i).$$

Further, we only need to know the length of the shortest path to each node in order to have a solution to the LP, not which nodes these paths goes through. Thus we omit storing the parent nodes. The new initialization becomes

1.)
$G.E := G.E \setminus \{(v_0, v_i) \mid i \in [1, n]\}$
$G.V := G.V \setminus \{v_0\}$
for each vertex $v_i \in G.V$
    $v_i.d = 0$.

We will need to adjust the input data a bit, so we can know which nodes of the graph that corresponds to variables with integer constraints. Name the vertices of the graph so that v_i is mapped from x_i for $i \in [1, n]$. Let the algorithm take in an additional list I of integers, corresponding to the indexes of the variables that are to be integral. We will interchange between referring to an integer vertex and an integer variable, where an integer vertex just means one mapped from an integer variable from the construction of the constraint graph. We will do the corresponding for the fractional case.

Rounding. In order to simplify computation, the easiest is to round down the relaxed upper bound at a node whenever the head of an edge is integral in general. This will happen whenever case 1 or case 4 is true. If both vertices of the edge, call it (x, y), is integral, that is, case 1 holds, then rounding down after calculating the estimate at y will give the same effect as rounding down the edge before calculating the estimate. Observe that

$$\lfloor c + f \rfloor = c + \lfloor f \rfloor, \quad \text{when } c \in \mathbb{Z}, \quad f \in \mathbb{R},$$

as f's fractional part will be the value rounded down in both cases. Substituting c with $x.d$, and f with $w(x, y)$, shows us that this is the case as long as $x.d \in \mathbb{Z}$. The tail, x, should already have an integer estimate from either the start value at 0, or from relaxing along another edge where x is the head. Since x is integral, any relax-operation with it as its head would result in a rounding into an integral value for $x.d$, since we construct the algorithm this way.

If only the head is integral, so that case 4 holds, rounding down the distance at the head, we get what is suggested online as a solution to the Cormen exercise. Why can we do this?

Any edge (v_i, v_j) with edge weight $w(v_i, v_j)$ comes from a constraint $x_j - x_i \leq b_k$. If v_j is an integral node, then it means x_j has to be integral in the LP. Equivalently, we have that

$$x_j \leq x_i + b_k,$$

where the right hand side may be fractional. By setting

$$x_j \leq \lfloor x_i + b_k \rfloor,$$

we do not cut away any feasible solution from the solution space, as this is the largest integral value x_j can take on. Translating this back into the graph, we correspondingly are allowed to end up with values given by

$$v_j.d \leq \lfloor v_i.d + w(v_i, v_j) \rfloor,$$

without removing any feasible solution of the original LP. As lower distance estimates always are updated as

$$v_j.d := \lfloor v_i.d + w(v_i, v_j) \rfloor,$$

it is consistent with the LP.

Further on, one would need to keep the rounded distance when propagating further through the graph, even though the upcoming vertices could have taken on a fractional value. To see this, assume that we round down an integer edge, (v_i, v_j), but continue with the fractional, greater, value when finding the distance to the next, fractional, vertex, v_k, via (v_j, v_k). Thus the path to v_j get some distance $\lfloor d \rfloor$, while the path to v_k would get the distance $d + w(v_j, v_k)$. The weight of the edge between v_j and v_k, $w(v_j, v_k) = b$, represent the maximum distance between these nodes. That is,

$$v_k - v_j \leq b \tag{4.5}$$

Had we not rounded when propagating, we would have gotten that

$$\begin{aligned} v_k - v_j &= (d + w(v_j, v_k)) - \lfloor d \rfloor && (4.6)\\ &= b + (d - \lfloor d \rfloor) \quad \text{where } (d - \lfloor d \rfloor) \in [0, 1) && (4.7)\\ \Rightarrow v_k - v_j &\geq b. && (4.8) \end{aligned}$$

Combining eq. (4.5) and eq. (4.8), we need to have that

$$\begin{aligned} v_k - v_j &= b \\ \Rightarrow d - \lfloor d \rfloor &= 0 \\ \Rightarrow d &= \lfloor d \rfloor. \end{aligned}$$

Thus rounding down relaxed values whenever the head is integral, and keeping these distances for further calculations of upper bounds on path lengths, is necessary.

Whenever the head of an edge has an index contained in the set I, we know this is an integer node. The relax operation of the altered algorithm will thus look like

2.)

if $j \in I$
  if $v_j.d > \lfloor v_i.d + w(v_i, v_j) \rfloor$
    $v_j.d := \lfloor v_i.d + w(v_i, v_j) \rfloor$
else
  if $v_j.d > v_i.d + w(v_i, v_j)$
    $v_j.d := v_i.d + w(v_i, v_j)$**.**

If relax operations stops updating, then the LP is feasible. Let us assume that the we can run the relax operations on all edges as many times we want. Every time a relax is made, one "fixes" a difference constraint. More precisely, one forces

$$\begin{aligned} & v_j.d \leq \lfloor v_i.d + w(v_i, v_j) \rfloor \\ \Leftrightarrow\ & x_j \leq \lfloor x_i + b_k \rfloor \quad \text{when } x_j \text{ integral, and} \\ & v_j.d \leq v_i.d + w(v_i, v_j) \\ \Leftrightarrow\ & x_j \leq x_i + b_k \quad \text{when } x_j \text{ fractional,} \end{aligned}$$

immediately after the relaxation of edge (v_i, v_j). This is easy to see, as either the relax operation did not update, and so the condition of the check did not hold, meaning that we need to have

$$v_j.d \leq \lfloor v_i.d + w(v_i, v_j) \rfloor \quad \text{or} \quad v_j.d \leq v_i.d + w(v_i, v_j),$$

or it did update, meaning that we afterwards have

$$v_j.d = \lfloor v_i.d + w(v_i, v_j) \rfloor \quad \text{or} \quad v_j.d = v_i.d + w(v_i, v_j).$$

In either case, the less or equal to condition is true.

One cannot guarantee that updating some values will not violate others. However, at the point when going through all the edges of the graph and no relax update is issued, we know that every inequality is fulfilled, and it is guaranteed that all the constraints of the LP is upheld. At this point the distances in the graph is guaranteed to be a solution. So, in other words, if the algorithm stops, we are guaranteed to have a solution.

On the other hand; if there exists a solution to the LP, will the algorithm stop? Theoretically one could imagine that there is a solution, but that the relax operations end up fluctuating the updated distances back and forth around a solution without converging. However, if the path-relaxation property holds even when rounding, then we know that if there exists a shortest from v_0 to any vertex v_i, then the algorithm will also converge at the distance. The proof that this holds is fairly similar to the proof for the original case of not rounding, and we provide the argument in appendix A.

When to conclude on infeasibility. At this point we know that if we allow the algorithm to run as long as the relax operations will update distances, then it will stop with an answer if and only if the LP is feasible. This is well and good, but when can we determine that there is no solution? One may believe it still is after $n-1$ iterations so that all simple paths are eventually relaxed (with possibly intermittent relaxations) in order. It turns out that is is not necessarily the case. Let us look at an example.

Figure 4.1 shows an example of a difference constraint graph where we imagine we have substituted the source v_0 and the edges $\{(v_0, v_i) | i \in [1, 6]\}$, all of weight 0, with upper bounds of 0 at all nodes, and use all nodes as sources. Thus it is ready for running the adjusted algorithm. We will show that there is a shortest path traversing 9 edges, so that the algorithm stops after maximum 9 iterations.

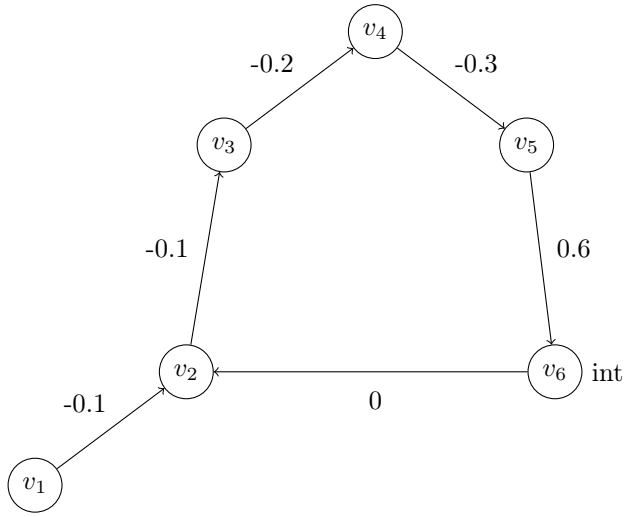

Figure 4.1: An example constraint graph where v_6 is integral, and the rest are fractional. v_0 and all its edges of distance 0 to the other nodes were discarded before running the algorithm. Here the relax operations stops updating after 9 iterations, even though there are only $n = 6$ original nodes.

The graph has the cycle

$$v_2 -> v_3 -> v_4 -> v_5 -> v_6 -> v_2.$$

Disregarding v_1 for a minute, starting from any point in the cycle and updating along its edges, we will converge at

$$\begin{aligned} v_2.d &= 0.0 \\ v_3.d &= -0.1 \\ v_4.d &= -0.3 \\ v_5.d &= -0.6 \\ v_6.d &= 0.0 \end{aligned}$$

We see that the cycle stops updating after maximum 3 iterations following the path from v_2 to v_5. As the integral offset when reaching v_6 ends up with being .0, and we do not round any fraction here, the cycle "settles" at summing up to 0.

But look at what happens if we follow the path from v_1. The first round, that is after maximum 5 iterations, we will have a negative .1 offset at v_6, and would round down to -1. We see this from following the path

$$v_1 -> v_2 -> v_3 -> v_4 -> v_5 -> v_6,$$

where we get

$$v_6.d = \lfloor -0.1 - 0.1 - 0.2 - 0.3 + 0.6 \rfloor = \lfloor -0.1 \rfloor = -1.$$

At this point, however, this value will propagate further through the cycle via maximum 4 more iterations,

$$v_6 -> v_2 -> v_3 -> v_4 -> v_5,$$

and give the same fractional offsets as for the single cycle for every node, but now moved down an integer. That is

$$\begin{aligned} v_2.d &= -1.0 \\ v_3.d &= -1.1 \\ v_4.d &= -1.3 \\ v_5.d &= -1.6 \\ v_6.d &= -1.0. \end{aligned}$$

When reaching v_6, the offset is .0, and it settles at -1.

$$v_6.d = \lfloor -1 + 0 - 0.1 - 0.2 - 0.3 + 0.6 \rfloor = \lfloor -1.0 \rfloor = -1.$$

Since no node on the cycle can improve its value further from here, and v_1 has no incoming edges, the algorithm stops at a solution:

$$\begin{aligned} v_1.d &= 0.0 \\ v_2.d &= -1.0 \\ v_3.d &= -1.1 \\ v_4.d &= -1.3 \\ v_5.d &= -1.6 \\ v_6.d &= -1.0. \end{aligned}$$

Since the maximum number of edges on a shortest path is 9, via the path

$$v_1 -> v_2 -> v_3 -> v_4 -> v_5 -> v_6 -> v_2 -> v_3 -> v_4 -> v_5,$$

we need 9 iterations to be sure that these edges are relaxed in order and a solution is found. Clearly

$$9 > (n-1) = 5,$$

where 5 should be the bound for the number of iterations needed to guarantee that any solution is found for the original Bellman-Ford. So we see from this example that this is not the case anymore.

The example above is very simple. It contains only one cycle, only one integer node, only one edge leading into the cycle, and none edges leading out. When a colleague of Hetland, Halvard Hummel, looked at the matter, he found another example where we could construct it to converge after n^2 iterations. This was a chain of multi-edged nodes, plus some nodes with incoming edges to this chain, constructed in a way so that the updates would fluctuate back and forth through this chain. This was another rather simple example, and so it is hard to predict what happens if we combine a great number of interconnected cycles. Can we still find an upper bound on the number of iterations needed to guarantee that no solution can be found?

For us it looks a bit too extensive to prove. A series of tests, where one runs the algorithm on several different generated problems to see if one can observe any significant patterns, or if the algorithm tends to terminate within a given time in relation to the graph structure, would possibly be one way to go. We restricted our scope to omit this for now, but it could be interesting to look at in the future.

Note that when running such problems with open source solvers, it seems that simple difference constraint problems often can be solved with preprocessing methods. A such, maybe other recognition procedures would prove useful. However, as we will see in chapter 5, for some cases our algorithm does indeed solve difference constraints faster.

So when should we stop searching when implementing the algorithm? One could use a limit on the time taken to run the algorithm, or a limit on the number of iterations, in order to break out of the algorithm if it does not terminate with an answer before this limit is reached. If we use the alterations from the pseudocode sections 1.) and 2.), together with a limit on the number of iterations, we get the following adjusted Bellman-Ford algorithm.

Algorithm 4 Adjusted Bellman-Ford

```
BELLMAN-FORD(G, w, s, I, limit)
 1: G.E := G.E \ {(v_0, v_i) | i ∈ [1, n]}
 2: G.V := G.V \ {v_0}
 3: for each vertex v_i ∈ G.V
 4:     v_i.d = 0
 5: changed = FALSE
 6: iterations = 0
 7: do
 8:     changed = FALSE
 9:     for each edge (v_i, v_j) ∈ G.E
10:         if j ∈ I
11:             if v_j.d > ⌊v_i.d + w(v_i, v_j)⌋
12:                 v_j.d := ⌊v_i.d + w(v_i, v_j)⌋
13:                 changed = TRUE
14:         else
15:             if v_j.d > v_i.d + w(v_i, v_j)
16:                 v_j.d := v_i.d + w(v_i, v_j)
17:                 changed = TRUE
18:     iterations += 1
19: while changed and iterations < limit
20: if changed
21:     return UNKNOWN
22: return TRUE
```

4.1.3 Moving the Solution

The output of algorithm 4 will always give solutions less than or equal to zero, because the initial upper bounds on path lengths are set to zero, and we only update when a better, decreased, path distance is found. If one would want all the variables of a solution to be greater than or equal to zero, it is easy to obtain this from the original output of the algorithm.

If the vector $\mathbf{x}^*$ is a solution to the linear program, we can see that $(\mathbf{x}^* + t)$, for any constant $t \in \mathbb{R}$ added to each element of $\mathbf{x}^*$, is also a solution. For any constraint

$$x_j - x_i \leq b_k,$$

we have that

$$x_j^* - x_i^* \leq b_k \quad \Rightarrow \quad (x_j^* + t) - (x_i^* + t) = x_j^* - x_i^* \leq b_k, \tag{4.9}$$

and thus it is possible to move all values of the solution by a constant term.

In fact, we can move the solution above or below any given constant. If we need to move the solution above a constant, we make sure that the lowest output is moved far enough so that all values are large enough. Similarly, when moving all variables below a constant, we make sure that the greatest output is moved far enough down.

More precisely, assume that we have a difference constraints problem where

$$x_i \geq c \quad \forall i \in [1, n], \tag{4.10}$$

were $c \in \mathbb{R}$ is a constant. Let x_k^* be the lowest real number of the solution $\mathbf{x}^* \in \mathbb{R}^n$ of the adjusted Bellman-Ford in algorithm 4. That is,

$$x_k^* \leq x_i^* \quad \forall x_i^* \in \mathbf{x}^*.$$

Find the minimum integer value, $t \in \mathbb{Z}$, we have to add to x_k^* in order for it to be greater than c, that is

$$t = \lceil c - x_k^* \rceil.$$

Let $\mathbf{x}'$ denote the vector we get by adding t to every entry of $\mathbf{x}^*$. Equation (4.9) assures that the new values of $\mathbf{x}'$ uphold the difference constraints. Further, we

see that the constraints of eq. (4.10) are all upheld, as

$$\begin{aligned} x'_i &= x^*_i + t \\ &= x^*_i + \lceil c - x^*_k \rceil \\ &\geq x^*_i + (c - x^*_k) \\ &= x^*_i - x^*_k + c \\ &= p + c \\ &\geq c \quad \forall i \in [1, n], \end{aligned}$$

because $p \geq 0$ follows from that x^*_k was a minimal value of the solution. Lastly, since the group of integers is closed under addition, by rounding t up before adding it to the solution, we preserve the integrality of the variables which are constrained to be integers.

We can do the case when all variables has to be less than a constant in a similar way.

The implementation for the case when all variables are required to be greater or equal to zero, could then look like

```
1:  if (Run algorithm 4) == true)    ∧   (x_i ≥ 0   ∀i ∈ [1, n]    is a requirement)
2:  min := 0
3:  for i = 1 to i = |G.V| − 1
4:        x_i := v_i.d
5:        if x_i < min
6:              min := x_i
7:        end
8:  end
9:  min := |min|
10: if (there exist integer constraints)
11:       min := ⌈min⌉
12: end
13: for i = 1 to i = |G.V| − 1
14:       x_i := x_i + |min|
15: end
```

Since $min = \min_i x_i$, and $x_i \leq 0 \ \forall i$, this would result in the minimum value gaining 0, and the other values ending up greater than or equal zero. If any variable is required to be integer, then the minimum variable value will be contained in the interval $[0, 1)$, succh that an integer is added to all variables of the solution output. This way we guarantee that the integer variables remain integer.

In other cases one might want some of the variables to be greater than zero, and some variables less than zero. In order to achieve that, one would need a solution where, when sorting the variable values in increasing order, one could cut the set in two, where the greatest valued set would be the variables required to be greater than zero. Then one can move the solution by a constant so that these end up being greater than zero, and the other half less.

We can enforce the split of variables by introducing several new inequalities to the LP, resulting in new corresponding mapped edges in the constraint graph. Let A be the set of variables required to be greater than or equal to zero, and let B be the set of variables required to be less than or equal to zero. By setting

$$\begin{aligned} x_j \leq x_i \qquad & \forall i \in A, \quad j \in B \\ \Leftrightarrow x_j - x_i \leq 0 \qquad & \forall i \in A, \quad j \in B, \end{aligned}$$

one enforce the sorted division of the two sets. If the problem is feasible, one can move all values of the solution so that zero falls in the middle of the two sets. Observe that it is allowed for variables at the boarder between the sets to be equal, as they are allowed to equal zero. If any variable is required to be integer, it is important that we make sure that it is possible to move the variables by an integer, while the sets of variables still are such that zero falls in the middle. If this is not possible, then the problem is infeasible.

We can, of course, generalize the reasoning above to hold for any bound, not only zero. In the implementation, we accepted any bound, as long as it was the same bound for every variable with a `less_than` or `greater_than` constraint. One may find solutions for simple cases of different bounds for different variables, but as for now we decided to only support up to one value for a bound.

Note that we add $|A| \cdot |B|$ new equations to the system, where $|A| + |B| = n$, when n is the number of variables. The most inequalities that can be added is then $(\frac{n}{2})^2$, since for any given perimeter the square maximizes the area. The upper bound on the new number of inequalities will be $[m + (\frac{n}{2})^2]$, so that the

run time will be

$$\mathcal{O}(n \cdot (m + (\frac{n}{2})^2)) = \mathcal{O}(nm + n^3) = \mathcal{O}(n^3).$$

The last equality comes from the fact that m is bounded by $O(n^2)$, which we argued under "Forms of Inequalities".

4.1.4 Implementations for Systems of Difference Constraints.

The implementation of difference constraints follows the flow of what was explained in section 3.1.2, while using the different algorithms in this chapter.

We made the `Algorithm` subtype `DifferenceConstraints`, and defined its struct as follows

Listing 4.1: Difference constraints.

```
struct DifferenceConstraints <: Algorithm
    limit::Union{UInt128, Nothing}
end
DifferenceConstraints() = DifferenceConstraints(nothing)
```

where `limit` is a number defined to be the number of iterations the Bellman-Ford should cycle around and relaxing edges before it gives up, since we have not found a proven limit for when we can conclude on infeasibility. If nothing is set, then the default limit will be set to $n - 1$, where n is the number of variables.

The code is structured into three files; `optimize.jl`, `recognize.jl`, and `bellman-ford-adjusted.jl`. The `optimize.jl` file contains the definition of

```
optimize!(model::AlgoModel,difference_constraint::DifferenceConstraints)
```

This function checks if the single variable bounds are supported, as all existing bounds should be equal for when moving the solution.

Then `recognize(model, difference_constraint)` is called, defined in `recognize.jl`, which returns a boolean indicated if the model contains difference constraints. We extended it to return `true` when a subset of constraints form difference constraints as well, since if it is infeasible, then we know the

whole problem has to be infeasible as well. Thus a vector of indices corresponding to the rows of `A` being difference constraints is returned, together with a possible updated vector `b`, in the case where the constraints are normalized. Further a graph is created via functions from the Julia packages `Graphs` and `SimpleWeightedGraphs`. The graph data, indices indicating which variables should be integers, and the iteration limit, is then sent to our implementation of `bellman_ford_adjusted()`, which is an implementation of algorithm 4. This implementation can be found in `bellman-ford-adjusted.jl`. This function returns **`true`** if the problem is feasible, **`false`** if it is infeasible, or throws an `CannotKnowError()`, which we created for this use. If the `optimize!()` function catches this error, it sets the termination status to `Trm_IterationLimit`, and solution status to `Sln_Unknown`.

Chapter 5

Practical Application and Tests

5.1 Experimental Research Questions

In order for a module of recognition procedures to be useful, it should provide faster answers for the specific problem classes than general open source solvers. Of course, the performance of such recognition procedures will be dependent on how they are implemented, but taken together with the preprocessing and building of `LPRep`, it is still interesting to look at the example for difference constraints.

Further, building the `LPRep` of an `AlgoModel` object could take some time. Since this has to be done in addition to the building a JuMP model, it would be interesting to see how long time this additional building would take by its own. A long building time may infer a need for a different type of preprocessing or storing of data.

When it comes to memory usage, the amount of memory used in `LPRep` is potentially big compared to what is used in a JuMP model. The JuMP model will be referenced by `AlgoModel`, not copied, so we need to consider the extra

memory usage induced by `LPRep`, and how it grows related to how the memory usage of a JuMP model grows. If the memory usage would grow exponentially as a function of the JuMP model's memory usage, for example, we need to consider other ways of referencing the problem data.

To sum up we have formulated three research questions:

- RQ1: Does our recognition procedure perform better in Julia than wrapped open source solvers when used on a subset of difference constraints?
- RQ2: How long time does our building and preprocessing of the model compared to building the same JuMP model?
- (RQ3: How much memory does `AlgoModel` use compared to a JuMP model?

5.2 Experimental Setup

For our tests we used a computer with 16 GB RAM and a Intel Core i5 processor with a 2,3 GHz Quad-Core. We used the `BenchmarkTools` package of JuMP in order to time different operations.

Because we ended up being pressed on time, we did not generate the amount of test problems we would like. However. we tried to diversify the problems and time each a large number of times in order to minimize noise by computer background processes. The problems we created, or downloaded, were divided into four categories.

- 17 small created problems, where all the constraints are difference constraints.
  - Of these are 10 feasible, and 7 infeasible.
  - These problems can be found in the `benchmarking/difference_constraints_problems` folder of the code base.
- 7 small created problems, where a subset of the constraints of each problem are difference constraints.

  - All of these are infeasible, because the subset of difference constraints are infeasible.
  - These problems can be found in the `benchmarking/difference_constraints_subset_infeasible` folder of the code base.

- 8 small created problems, where all have an inconsistency that AlgoModel can detect with its preprocessing routine.
  - These problems can be found in the `benchmarking/inconsistent_problems` folder of the code base.

- 20 of the first benchmark problems of MIPLIB, taken in alphabetical order.
  - These problems can be found on MIPLIB's web page, in the MIPLIB2017 benchmark set.

he two first sets are divided since solvers' presolving routines turned out to solve difference constraints quite easy. Thus we wanted to see if recognizing subsets of inconsistent difference constraints would be harder for the solvers than pure difference constraints or not.

When we ran problems with the GLPK solver, it kept running without terminatkng, even when we tried to exit the execution, so we had to force quit the process. This happened for example when running `benchmarking/difference_constraints_problems/infeasible_1.mps` in our project repository. Even if we sat a time limit on the optimizer this happened, so we suspect it could be some bug when it comes to JuMP and GLPK communicating. Thus we ended up doing the tests with only HiGHS, Cbc, and SCIP.

For each expression we wanted to benchmark, we ran it 1000 times, and then averaged the time and memory usage over these samples. That way we can eliminate noise from background processes on the computer. Since the building of the MIPLIB problems take a long time, we reduced this number to 10 times per run. Setting this parameter is done through setting the value of

```
BenchmarkTools.DEFAULT_PARAMETERS.samples
```

to the desired number.

Further, a `BenchmarkGroup` object was made for each problem set. We looped through the set in order to register every line of code which should be benchmarked, then we tuned the set, and at this stage we let `BenchmarkTool` decide the benchmarking parameters it deemed reasonable. Note that the number of samples per code line may be set to something different than 1000 at this stage due to the tuning process, though after inspection, it was not altered for most trials.

Whenever we wanted to exclude the lookup time of an argument of a function, we added a `$` in front of the corresponding argument. We did this when calling `optimize!()` on created JuMP models, or `AlgoModel` models, so that only the optimization was measured.

We also inspected (not in the code base) that all the results were equal, in the sense that only correct answers were recorded from all the methods of solving the LP's.

5.3 Results

Because of the small sample size, the results may not be as reliable as one could hope, so further work on more recognition procedures, and generation of a larger set of test problems, would give even more insight. However these tests give us a feel of how our module performs on certain problems. We decided to display various scatter plots to get a view of the trends. The tests that were run can be found in `benchmarking/benchmark.jl`.

5.3.1 RQ1

In fig. 5.1 we can see that for difference constraints, SCIP seems to be consistently fast, while the others vary a bit, though HiGHS also perform reasonably fast. This seems to be consistent with other sources, especially concerning SCIP. We see that `AlgoModel` seem to solve some problems almost as fast as SCIP, while for others it takes longer time than any other solver, so the speed seem to depend a lot on the specific problem in question. However, compared to Cbc, the worst run times of `AlgoModel` are in general less than the worst cases of Cbc,

except the one spike at problem of index 11.

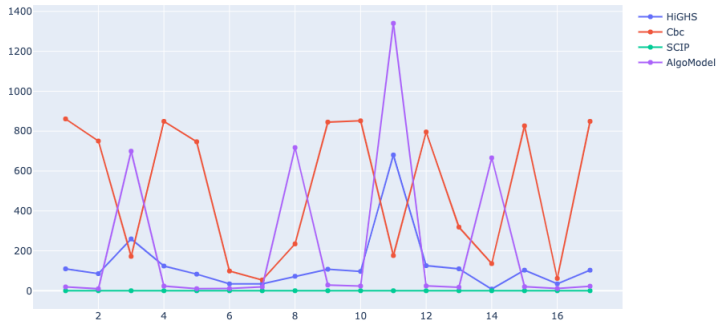

Figure 5.1: Running time for the first set of benchmarks, with a mix of feasible and infeasible difference constraints.
X-axis: problem index. Y-axis: microseconds.

Since we suspected that detecting difference constraints subproblems within the LP, and concluding on infeasibility whenever a subproblem is infeasible, is harder than constraints being pure difference cobnstraints, we tested some of these instances as well. However, as fig. 5.2 shows, the behaviors seems to be the same as in fig. 5.1, namely, `AlgoModel` fluctuating between close to SCIP and worst cases of Cbc. Why the runtime increased from sample 4 and on could be due to an interfering factor, for example increased background processes or memory use, or it could simply be related to the problems tested. Since the different tests are run separately, it could be that some factor only affected the time when `AlgoModel` ran, but since 1000 samples was run per reported value, and fig. 5.1 shows no similar trends of longer times towards the end of processing, it is as likely it is just the nature of the problems. So in general it seems like a recognition procedure could compete with a solver for some problems, and so a parallel procedure might be useful, though it has to be weighed against the additional time taken to build the `AlgoModel` object.

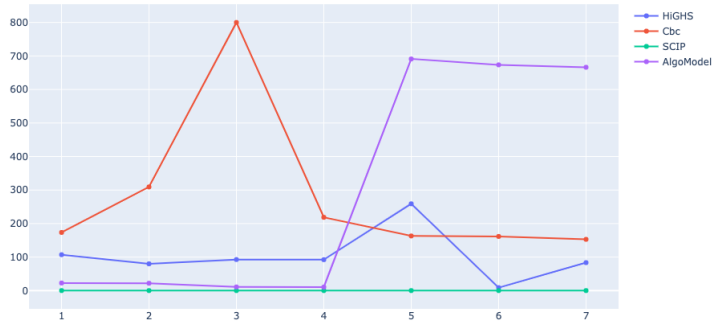

Figure 5.2: Run time for the second set of benchmarks, with infeasible problems due to a subset of infeasible difference constraints.
X-axis: problem index. Y-axis: microseconds.

5.3.2 RQ2 and RQ3

When it comes to building the models, we timed

```
MOI.copy_to(Model(), $file_model)
```

and

```
AlgoModel($model, DifferenceConstraints())
```

where `file_model` is a `MOI.Utilities.GenericModel` object which is an intermediate representation of the model when read from file. We do not time this reading, as it would have to be done for any model creation. We rather time the creation of an empty JuMP model, as the first argument of `copy_to()`, and the initialization of this model with the data of `file_model`, which is done with the `MOI.copy_to()` function. We compare this to the creation of an `AlgoModel` object with `DifferenceConstraints()` as a recognition procedure. The first argument, `model`, is an already created JuMP model. We do not time the lookup of the arguments containing the data, as denoted with a `$` preceding these arguments.

From fig. 5.3 through fig. 5.6, we see that both time, memory, and allocations

for creating an `AlgoModel` vs a JuMP `Model` are related in a similar way. It seems that even very big problems, where

$$m + n > 160000,$$

there can be instances where the creation of a JuMP model and a `AlgoModel` costs roughly the same amount of time and resources (as indicated by the second to last red dot on fig. 5.3, where it actually overlaps the blue dot). However, this means that the time needed to use `AlgoModel` will at best double the time of using just a JuMP model.

In fig. 5.6 we mapped allocation against problem index as well, in order to see if there are any reason for some model creations to take longer time. In fact, the spike of allocation in the tenth problem turns out to correspond to the creation of `bab2.mps`, which we already have encountered issues with, as discussed in section 3.1.4, where the substitution of fixed variables caused floating point precision issues. It is thus natural that these problems takes longer to build, as the preprocessing procedures of substituting variables is running. This shows that if the problem is so that substituting variables is necessary, possibly in several passes, it will take longer time to construct the inner `LPRep` representation.

It is difficult to land on a definite decision about how good the design of the module is with this data, because faster recognition procedures may justify the extra building time. When SCIP almost always performs better than `DifferenceConstraints()`, it is better to only use a JuMP model with SCIP attached.

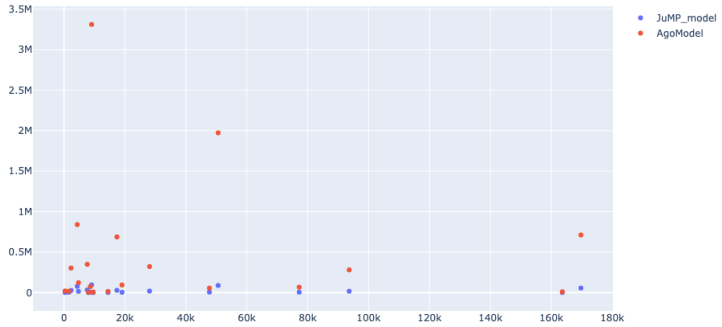

Figure 5.3: Time it takes to build a model, compared to the combined number of variables and constraints in the model. X-axis: problem size. Y-axis: microseconds.

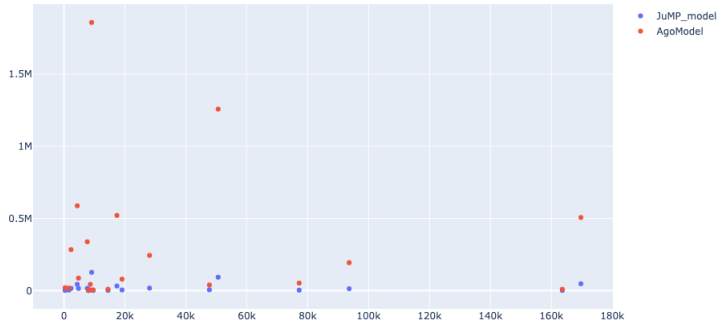

Figure 5.4: Memory usage given problem size.
X-axis: problem size. Y-axis: kilobytes.

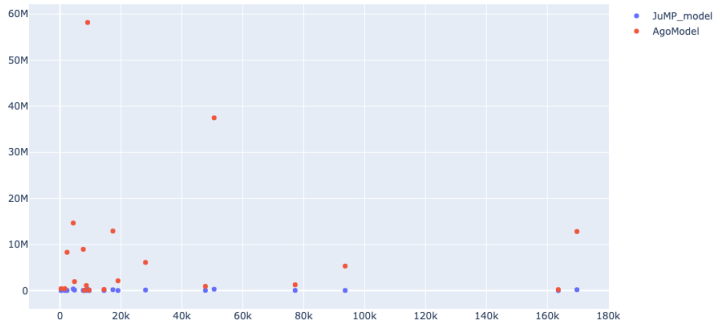

Figure 5.5: Allocations vs problem size.
X-axis: problem size. Y-axis: alocations.

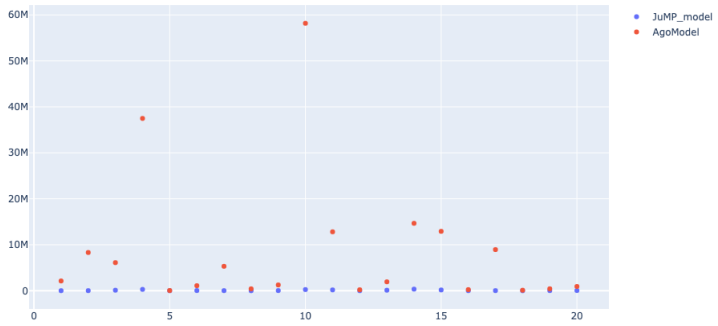

Figure 5.6: Allocations vs problem index.
X-axis: problem index. Y-axis: allocations.

5.3.3 Preprocessing

Comparing simple inconsistency checks, namely the ones we can detect during the building of the `AlgoModel`, would be interesting to time and compare against other solvers on the same problems. However, as this preprocessing is strongly tied to the building of the model, it is difficult to extract out and only time the preprocessing part of the code.

If we compare the time when calling `optimize!()` only, we see, in fig. 5.7, that it is very fast. This is as expected, as it narrows down to checking a flag `is_coinsistent`. SCIP is almost as fast, as shown in fig. 5.8, which is gained from zooming in on fig. 5.7. It could be interesting to figure out when SCIP does its preprocessing as well, and see if the process of attaching `SCIP.Optimizer` to a model, through `set_optimizer()`, would add any extra time compared to HiGHS and Cbc, in case of early preprocessing at initialization. Of course, since we in general not have inspected how and when the different solvers do their preprocessing, it is difficult to compare and comment on this data. Though it is interesting to see how SCIP seems to be exceptionally fast on most tests.

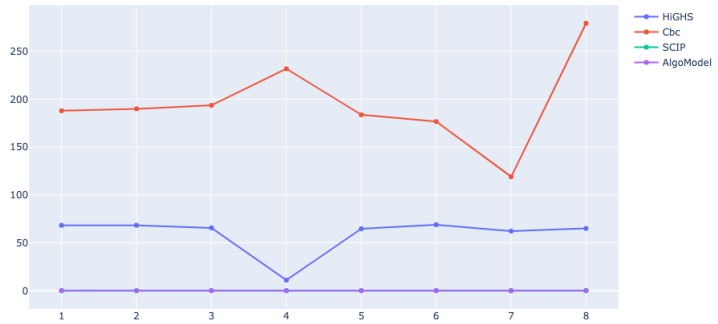

Figure 5.7: Time taken to detect simple inconsistencies through `optimize!()`. X-axis: problem index. Y-axis: microseconds.

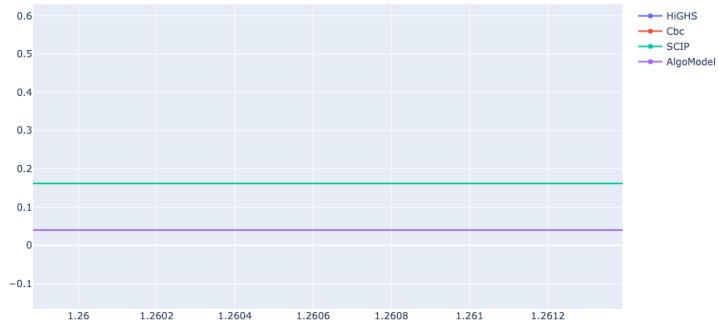

Figure 5.8: Time taken to detect simple inconsistencies through `optimize!()`. Zoomed in.
X-axis: problem index. Y-axis: microseconds.

Chapter 6

Discussion and Future Work

As we have seen, it seems like a module such as MipFlex could be helpful for certain problem classes, and so introducing more recognition procedures, and making the parallel component more robust, could have a great potential.

There exist interesting articles to be explored for implementing new recognition procedures. One such is on how to convert LP's to network problems[8] in $\mathcal{O}(rn)$ time, where r is the number of rows, and n is the number of nonzero entries of A. Since most network problems can be solved in polynomial time, such a transformation would be beneficial for sparse constraint matrices.

Another idea is to let an `Algorithm` object take in special information about the problem from the user, for example that the problem is a maximum flow problem. In that case, the algorithm does not have to recognize the problem, but can start applying Ford-Fulkerson right away. However, this may be a naive example, as going through the creation of an `Algorithm` and JuMP model would be unnecessary and induce extra work. However, the idea of sending known information or heuristics through `Algorithm` objects is an interesting idea to explore.

When it comes to the implementation, several improvements could be made. The obvious one is to make sure that running the procedures in parallel never will incur race conditions. In addition, returning whenever a solution is found,

instead of waiting for other threads to finish, is a crucial requirement for a successful use of parallelity. There are several other forms of possible enhancements as well. One could include solution data concerning the dual of the LP. One could implement prioritizing flags for each `Algorithm` object, so that when the code is run sequential, the module knows in which order to call `optimize!()` on the algorithms attached. One could look into when and how solvers do preprocessing, and improved our implementation on this aspect. And of course, more types of constraints, especially variable constraints could be supported. For sustainability, more get-er functions should be provided for `Algorithm` developers. If one also could support incremental changes to the model, not having to build the `LPRep` from scratch every time the JuMP model is updated, processing time could be saved.

Wrapping the module within a solver interface would make it easier to read the problem data, in-memory, without constructing an internal copy of the data, which in turn would typically half the time of model construction. The only reason for not doing this is the warning from JuMP of how big a task this is. However, we believe a wrapped MipFlex would improve both the ease of use, as well as the speed of the module. The JuMP interface is build, after all, to work with fast solvers, and not be a bottleneck in the process. Thus we believe that a natural step forward could be to write a JuMP wrapper for the functionality of a module like MipFlex.

Appendix A

Adjusted Bellman-Ford terminates iff LP feasible.

A.0.1 Disclaimer

This section is not cleaned up, as we did not manage to conclude our reasoning. Appendix A is only included as a possible help for others that may want to prove that the adjusted Bellman-Ford algorithm in algorithm 4 actual will terminate if there exists a solution, and if there exists an upper bound on the number of iterations.

A.0.2 Incomplete Sketch of a Possible Proof.

We will here focus on providing a more formal argument for the statement that if the the LP is feasible, then the algorithm will eventually return TRUE.

Proposition A.0.1. *Let* $\mathbf{Ax} \leq \mathbf{b}$ *be a system of difference constraints over the field* $\mathbb{R}$*, with* n *variables and* m *constraints. Assume an arbitrary amount of variables are constrained to be integers, and let* I *denote the set of indices of these integral variables. Let* $G = (V, E)$ *be the LP's corresponding constraint*

graph.

When running the tweaked Bellman-Ford on G, (ref adjusted bellman ford), with a theoretically infinite limit on iterations, it will terminate with a shortest path to each vertex if the original LP has a feasible solution. Shortest path, in this context, is defined to be the smallest accumulated distance from one vertex to another, following edges in the graph and rounding down to nearest integer whenever a vertex is integral.

Moreover, if the algorithm terminates with the return value $TRUE$, the final distance estimates will be shortest paths, and each shortest path will be a solution to the LP. Specifically we have that

$$\mathbf{x} = [v_1.d, v_2.d, \ldots, v_n.d]$$

will be a solution to the linear system of difference constraints.

Proof. We need to show that

The algorithm terminates with $TRUE \iff$ The LP is feasible.

($\Longrightarrow$)
We already showed, in section 4.1.2, "**If relax operations stops updating, then the LP is feasible**", that if the algorithm returns $TRUE$, then the final distance estimates make up a solution to the LP. Thus we concern ourselves with the other direction of the proof here.

($\Longleftarrow$)
For the other direction we will divide this into two parts. First, if the LP is feasible, then there exists shortest paths to all vertices in the constraint graph. Secondly, if there exists shortest distances, then the algorithm will fins these and return TRUE. So we have

1.) LP is feasible $\Longrightarrow$ All shortest paths exists.
2.) All shortest paths exists. $\Longrightarrow$ algorithm will terminate with TRUE.

1.) LP is feasible $\Longrightarrow$ All shortest paths exists.

Assume LP is feasible but that not all shortest paths exists in the corresponding constraint graph. Firstly, all nodes have distance 0 from the start node v_0 at initiation, so the only way for a path not to exist, is for it to be unbounded from below, going towards an infinite negative value.

Assume first that there are no cycles in the graph, and that v_k is a node with no shortest path. If the graph has no cycles, look at all paths from v_0 to v_k. This is possible because we have a finite number of nodes and edges, and any of these paths will not come back with a better estimate, as there are no cycles. Pick the minimum value of the rounded distance of these paths. We know that the path $p : v_0 -> v_k$ with weigh 0 exists, so there will be at least one value in the set. The minimum value will be a shortest path, and so v_k must have a shortest path.

Thus there has to be a cycle. Again assume v_k has no shortest path. If there are only simple paths with no cycles to it, we have the same situation as with the no cycle graph, so then assume there is at least one cycle on at least one of the paths to v_k.

Start by assuming that there is only one cycle in the graph. We will generalize with any number of cycles eventually.

v_k has to be either in the cycle, or outside of it. First assume v_k is in the cycle. Now, if following the cycle one round would result in a greater value when arriving at v_k, we can always skip the cycle. That is, if v_k is on the cycle, follow the path until we reach v_k, and do not follow the cycle, as it would only increase the value.

TODO: Insert image and explain better? TODO: Maybe distinctly name nodes v, and variables of LP x. If v_k is not in the cycle, the cycle has to be on the path between v_0 and v_k (else it would be a simple path from v_0 to v_k). Let v_e be the first node on the path that is a part of the cycle, and let v_s be the last node on the cycle before a simple path leads further to v_k. If we just use the path $p : v_0 ->' simplepath' -> v_e -> \cdots -> v_s ->' simplepath' -> v_k$, then it is a simple path with a finite weight, so assume we use the cycle at least once. It means we need to go a round from v_s back to v_s, so that we can re-emerge out from the simple path to v_k. If the length at v_s after rounding and following the cycle is greater than the first time we entered v_s, then this greater weight will either not affect v_k, or accumulate and result in a greater value at

v_k than if we had not followed the cycle. It could not result in a lesser value by adding the same edges of the exiting simple path, since the rounding down cannot result in a lesser value when rounding down from a greater value. Thus we can here as well "skip" the cycle.

Observe that if we follow a cycle once and obtaining a greater value, that is $v_k.d1 \leq v_k.d2$, where $v_k.d1$ is the distance at a node on the cycle without having followed the cycle, and $v_k.d2$ is after following it one round, then following it another round would result in a higher or equal value. That is, adding the same values and rounding down at the same instances cannot result in any lower number. Thus following the cycle more than once, when the first round does not provide any improvement on the path length, would not work either.

Thus it has to be the case that when following the cycle it leads to a lesser value in the path estimate, so that it would be beneficial to follow the cycle.

As we have demonstrated (ref fig), it is possible to follow a cycle once, gaining a lesser value, and then it stabilizes. If this happens, then the path is finite. So in order for it to be unbounded from below, we have to have the case that the path value decreases strictly *every single round*.

Assume that given any intermediate value of a path on the cycle, following it one round yields a strictly more negative value. Then, disregarding how we entered the cycle and the values of incoming paths, following the cycle would be beneficial. But then, since we know following the cycle from any value yields a more negative value, then following it another round would be beneficial, and so, for any path entering the cycle, it would be beneficial to keep following the cycle because it would guarantee a more negative value. Thus, every path with the cycle as a part of it would be unbounded from below. So without loss of generality, assume v_k is on the cycle. In particular, whatever value we have at v_k at any moment on any path, say $v.d$, after following one round, we will have that

$$v.d2 < v.d,$$

where $v.d2$ is the value after updating along the cycle.

TODO: Argue about weight of cycle.

Now, let us observe all edges of the cycle in the corresponding LP formulation. This corresponds to all inequalities $v_j - v_i \leq b_k$, where (v_i, v_j) is an edge

in the cycle. Since a cycle starts and ends at the same node, all nodes will have an even number of in and out edges. That is, every time we enter a node, we will also leave it, and it is true for the start node, as we start by leaving it, and enter it when there has been a cycle. Thus, by summing up the left hand side of the inequalities of the cycle

$$v_k -> v_{k+1} -> \cdots -> v_{k-1} -> v_k,$$

we get

$$(v_{k+1} - v_k) + (v_{k+2} - v_{k+1}) + \cdots + (v_k - v_{k-1}) = 0,$$

as nodes gets opposite signs when being at a head and a tail.

For the proof of the non-rounding case, we summed the right hand side as well, to find a bound on the edge sum of the cycle, that will eventually lead to a contradiction. In order to do it here, we need to tighten the bound of each inequality. Take a look at the following.

If both variables v_j and v_i of the inequality are integers, then the difference cannot take a value between b_k and $\lfloor b_k \rfloor$, and so

$$(v_j^{int} - v_i^{int}) \leq b_k \iff (v_j^{int} - v_i^{int}) \leq \lfloor b_k \rfloor,$$

where we denote a variable with the superscript of *int* whenever it is constrained to be integer.

Now if only one of the variables in an inequality is constrained to be integer, then we have two different cases. Let first v_i^{int} be the integer variable. Define

$$F(x) = |x| - \lfloor\, |x| \,\rfloor, \quad \in \mathcal{R},$$

where $|x|$ is the absolute value operator. $F(x)$ then returns the fractional part of x. If the numerical value of v_j and $(v_j - v_i^{int})$ have both the same sign (that is both positive or both negative), then the difference of the two numbers has to be $F(v_j)$. If they have opposite signs, the fraction goes "via" zero, and so an integer number added or subtracted will be left with $(1 - F(v_j))$.

2.) All shortest paths exists. $\implies$ algorithm will terminate with TRUE.
We will prove several properties and combine them into an argument for that if all the shortest paths exists in the constraint graph, then the algorithm will

terminate upon them and return TRUE. Since these properties are already proved (ref Cormen) with fractional vertices, we will concern ourselves with cases of integral vertices.

Triangle inequality. The triangle inequality for these types of shortest paths will follow the same argument as for Bellman-Ford. That is, if there is a shortest path $p_{0,j}$ from vertex v_0 to v_j, where v_j is integral, then by definition of "shortest", no matter what other path $p'_{0,j}$ from v_0 to v_j you find, possibly rounding down a different factor on some of the same edges of $p_{0,j}$, the distance will be as great as, or greater than, $p_{0,j}$, else it would not be a shortest path. In particular, if $p'_{0,j}$ is starting with the shortest path to another vertex v_i, and then traversing (v_i, v_j) we will have that

$$\delta^*(v_j) \leq \lfloor \delta^*(v_i) + w(v_i, v_j) \rfloor. \tag{A.1}$$

Upper-bound property. The Upper-bound property states that during the execution of the algorithm, $v.d \geq \delta^*(v)$ for any vertex v, and that when $v.d = \delta^*(v)$, it will never change. We know that there exist paths to all vertices of distance 0 from v_0, so any shortest path cannot be more than this. Thus, in the beginning, $v.d = 0 \geq \delta^*(v)$ for all vertices v. Since v_0 has no edges going into it, the distance to v_0 cannot be improved by any path, and so we know that $\delta^*(v_0) = 0$ from the start. By induction, assume that at some step during the algorithm we update the d attribute of some vertex v. For not repeating proofs, assume this is an integral vertex. From the algorithm, the updated value will be given as

$$v.d = \lfloor u.d + w(u, v) \rfloor,$$

via an edge from another vertex u. By the inductive hypothesis, we know that $u.d \geq \delta^*(u)$, so

$$v.d \geq \lfloor \delta^*(u) + w(u, v) \rfloor.$$

By the triangle inequality the right hand side is greater or equal to $\delta^*(v)$, and so we end up with

$$v.d \geq \delta^*(v),$$

and so by induction the upper bound holds in all cases. From definition of the algorithm, the update of d can only decrease its value, and so, when a shortest

path value is found, it cannot decrease further since $v.d \geq \delta^*(v)$. Thus, once $v.d = \delta^*(v)$, $v.d$ will never change its value.

Existence of shortest paths made up of shortest paths. Note that in the original proof of Belmann-Ford, one proves that subp-aths of shortest paths are shortest paths. However, this is not necessarily true with the rounding case, as rounding can make a sub-path of greater length end up with the same distance to a specific edge, as Figure A.1 shows. Here both the paths

$$\begin{aligned} p_{0,k}: &\quad v_0 \rightsquigarrow v_i \rightsquigarrow v_j \rightsquigarrow v_k \\ p'_{0,k}: &\quad v_0 \rightsquigarrow v_i \rightsquigarrow v_a \rightsquigarrow v_j \rightsquigarrow v_k \end{aligned}$$

will be shortest path from v_0 to v_k, as both determines the weight -1 because of rounding:

$$\begin{aligned} p_{0,k}: &\quad d(v_k) = \lfloor 0 + 0 + (-0.1) \rfloor = -1 \\ p'_{0,k}: &\quad d(v_k) = \lfloor 0 + (-0.1) + (-0.8) + (-0.1) \rfloor = -1. \end{aligned}$$

Thus $p_{0,k}$ is a shortest path, but the sub-path of $p_{0,k}$

$$p_{i,j} : v_i \rightsquigarrow v_j, \quad \text{with weight} \quad d(p_{i,j}) = 0,$$

is not the shortest path from v_i to v_j, as

$$p'_{i,j} : v_i \rightsquigarrow v_a \rightsquigarrow v_j, \quad \text{with weight} \quad d(p'_{i,j}) = -0.1 + (-0.8) = -0.9$$

is shorter. However, as long as we can show that there exists at least one shortest path $p_{0,k}$, for any v_k, which is such that any other node v_x on this path gets its shortest distance from v_0 by using the same path $p_{0,k}$ up until v_x, we can use this property further in the proof.

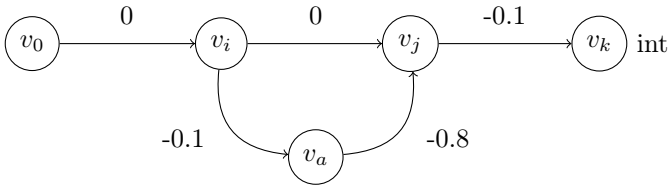

Figure A.1: Sub-paths of shorter paths need not be shortest paths. We disregard all the other 0 weighted edges from v_0 for simplicity.

To see why this is true, take any vertex v_i, and look at all its in-edges. Let $V_i = \{u \in G.V \mid (u, v_i) \in G.E\}$, that is, all vertices at a tail of any in-edge to v_i. To minimize distance to v_i, the best we can achieve can be written as

$$\begin{aligned} &\min_u\{\delta^*(u) + w(u, v_i) \mid u \in V_i\} \quad \text{if } v_i \text{ fractional} \\ &\min_u\{\lfloor\delta^*(u) + w(u, v_i)\rfloor \mid u \in V_i\} \quad \text{if } v_i \text{ integral,} \end{aligned}$$

as using something greater than the shortest distance to any neighboring vertex can only make the final sum equal or greater to what we achieved above. The important aspect of the integral case is that if we have $d(u) > \delta^*(u)$, where $d(u), \delta^*(u) \in \mathbb{R}$, then also $[d(u) + w(u, v_i)] > [\delta^*(u) + w(u, v_i)]$, where $w(u, v_i) \in \mathbb{R}$, and when rounding down these two real numbers, where one is greater than the other, we can never end up with $[d(u) + w(u, v_i)] < [\delta^*(u) + w(u, v_i)]$. The "best" we can achieve is equality.

Let v_j be the tail of one of the edges resulting in a shortest distance to v_i. In (ref) we use the shortest distance to v_j, so let us find this. The shortest distance to v_j can be expressed as

$$\begin{aligned} &\min_u\{\delta^*(u) + w(u, v_j) \mid u \in V_j\} \quad \text{if } v_j \text{ fractional} \\ &\min_u\{\lfloor\delta^*(u) + w(u, v_j)\rfloor \mid u \in V_j\} \quad \text{if } v_j \text{ integral,} \end{aligned}$$

where V_j is defined in a similar way as V_i. Again choose v_k which is the tail of one edge giving the minimum distance to v_j.

Since we know that shortest paths exists, they cannot be infinite, and so a final number of c vertices will make up the path to v_i. Further on, all vertices are reachable from v_0 from construction. Thus we can continue in this fashion until we end up with v_0 as a tail vertex, which we argued has a shortest distance of value 0. Using these vertices, in the opposite order as we picked them, will construct a path p from v_0 to v_i, where each vertex on this path gain its shortest distance from v_0 via it.

Property after relaxing. Observe that immediately after relaxing a rounding edge $(u, v) \in E$, we have

$$v.d \leq \lfloor u.d + w(u, v)\rfloor.$$

That is, either we updated $v.d$ so that $v.d = \lfloor u.d + w(u, v)\rfloor$, or it did not update, meaning we must have had $v.d < \lfloor u.d + w(u, v)\rfloor$. We will use this to show the convergence property for rounding edges.

The convergence property. The convergence property states that if we know that there is a shortest path from v_0 to v where the last edge on this path is (u, v), denoted by $v_0 \rightsquigarrow u \to v$, then if $u.d = \delta^*(u)$ prior to a relax call, then $v.d = \delta^*(v)$ at all times after the call.

First off, since we know there is a path from v_0 to v, then due to (ref "subpath" property), we know we can choose a path where every subpath from v_0 is a shortest path. Choose such a path, p, and let u be the vertex prior to v on p. Assume $u.d = \delta^*(u)$ at some point during the algorithm. Then the upper-bound property secures that this will stay the same for the rest of the execution. Let us assume the last edge, (u, v) is a rounding edge from a fractional to an integral vertex. Then, after relaxing (u, v), we have from (ref prev prop) that immediately after this relaxation,

$$\begin{aligned} v.d &\leq \lfloor u.d + w(u, v) \rfloor \\ &= \lfloor \delta^*(u) + w(u, v) \rfloor \\ &= \delta^*(v), \end{aligned}$$

where the last comes from the fact that this path is constituted from shortest paths. However, by the upper-bound property we have $v.d \geq \delta^*(v)$, and so we must have that

$$v.d = \delta^*(v).$$

The upper-bound property also ensures that this equality is maintained during the rest of the execution of the algorithm.

The path-relaxation property. The path-relaxation property guarantees our claim, that all vertices receives δ^* at some point, and that the algorithm then returns TRUE. This follows almost directly from the upper-bound property(ref), the convergence property(ref), and existence of shortest path made up of shortest paths(ref). If there is a shortest path from v_0 to v_k, then (ref) guarantees that there exists a path $p =< v_0, v_1, \ldots, v_k >$ where every vertex on p also obtains its shortest distance with rounding, δ^*, on this path. If a sequence of relaxation steps relaxes all edges on this path, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, in order, then $v_k.d = \delta^*(v)$ after these steps and at all times after. To see this, induction was used in Cormen(ref), and can be used here as well. For initialization, $d.v_0 = 0$. This is guaranteed to be minimized because of no incoming edges, and by the upper-bound property it will stay the same during execution. If we at any step have $v_{i-1}.d = \delta^*(v_{i-1})$, and we relax edge (v_{i-1}, v_i), then by the convergence property $v_i.d = \delta^*(v_i)$ after relaxation and at all times later.

Since we know that there exits finite shortest paths to every vertex, we are guaranteed that after the maximum length path, all vertexes have gained their shortest path value. At this point, no further relax operation will be issued, because

$$v_j.d = \delta^*(v_j) \leq \lfloor v_i.d + w(v_i, v_j) \rfloor, \quad \text{if } v_j \text{ integral}$$
$$v_j.d = \delta^*(v_j) \leq v_i.d + w(v_i, v_j), \quad \text{if } v_j \text{ fractional},$$

and so $CHANGED = FALSE$ in (ref algo), *iterations* will be less than *limit* since we assumed to have infinite iterations available, and so the algorithm will return $TRUE$.

□

Bibliography

[1] H. et. al. "Parallelizing the dual revised simplex method". In: *Mathematical Programming Computation* 10 (2018), pp. 119–142. DOI: https://doi.org/10.1007/s12532-017-0130-5.

[2] A. B. et al. "Complexity of branch-and-bound and cutting planes in mixed-integer optimization". In: (Nov. 2020). DOI: https://doi.org/10.48550/arXiv.2003.05023.

[3] H. Andersen. *MipFlex*. 2023. URL: https://github.com/henriean/mip-flex (visited on 02/08/2023).

[4] M. F. Anjos, A. Lodi, and M. Tanneau. *Tulip.jl: an open-source interior-point linear optimization solver with abstract linear algebra*. Tech. rep. 2019. URL: https://www.gerad.ca/fr/papers/G-2019-36.

[5] H. I. Arvidsen. *Hydropower Production Scheduling using Stochastic Dual Dynamic Programmingsubject to environmental constraints*. Norway: University of Bergen, 2019. URL: https://bora.uib.no/bora-xmlui/bitstream/handle/1956/20360/masterThesis_HIA.pdf?sequence=1&isAllowed=y (visited on 01/07/2021).

[6] T. Berthold. "Primal Heuristics for Mixed Integer Programs". In: (Sept. 2006).

[7] K. Bestuzheva et al. *The SCIP Optimization Suite 8.0*. Technical Report. Optimization Online, Dec. 2021. URL: http://www.optimization-online.org/DB_HTML/2021/12/8728.html.

[8] R. E. Bixby and W. H. Cunningham. "Converting Linear Programs to Network Problems". In: (Aug. 1980). URL: https://www.jstor.org/stable/3689441.

[9] COIN-OR Foundation. *COIN-OR Branch-and-Cut solver*. 2022. DOI: 10.5281/zenodo.6522795. URL: https://github.com/coin-or/Cbc (visited on 02/08/2023).

[10] T. H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[11] I. I. Cplex. "V12. 1: User's Manual for CPLEX". In: *International Business Machines Corporation* 46.53 (2009), p. 157.

[12] G. B. Dantzig. *Origins of the Simplex Method*. Stanford University: Department of Operations Research, 1987. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a182708.pdf (visited on 01/08/2021).

[13] I. Dunning, J. Huchette, and M. Lubin. "JuMP: A Modeling Language for Mathematical Optimization". In: *SIAM Review* 59.2 (2017), pp. 295–320. DOI: 10.1137/15M1020575.

[14] A. Gleixner et al. "MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library". In: *Mathematical Programming Computation* (2021). DOI: 10.1007/s12532-020-00194-3. URL: https://doi.org/10.1007/s12532-020-00194-3.

[15] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2023. URL: https://www.gurobi.com.

[16] P. S. HD Mittelmann. "Decision tree for optimization software". In: (2005). URL: http://plato.asu.edu/bench.html.

[17] B. Legat et al. "MathOptInterface: a data structure for mathematical optimization problems". In: *INFORMS Journal on Computing* (2021). DOI: 10.1287/ijoc.2021.1067.

[18] R. Lougee-Heimer. "The common optimization INterface for operations research: Promoting open-source software in the operations research community". English. In: *IBM Journal of Research and Development* 47.1 (Jan. 2003). Copyright - Copyright International Business Machines Corporation Jan 2003; Last updated - 2022-10-23; CODEN - IBMJAE, pp. 57–66. URL: https://www.proquest.com/scholarly-journals/common-optimization-interface-operations-research/docview/220690110/se-2.

[19] H. Mittelmann. *MIPLIB17 benchmark*. 2005. URL: http://plato.asu.edu/ftp/milp_tables/8threads.res (visited on 02/08/2023).

[20] H. Mittelmann. *MIPLIB17 benchmark summary*. 2005. URL: http://plato.asu.edu/ftp/milp.html (visited on 02/08/2023).

[21] E. Oki. “GLPK (GNU Linear Programming Kit)”. In: 2012.

[22] A. D. Pia and S. D. Gregorio. “On the complexity of binary polynomial optimization over acyclic hypergraphs”. In: (14 2022). DOI: https://doi.org/10.48550/arXiv.2007.05861.

[23] M. Savelsbergh. “Preprocessing and Probing Techniques for Mixed Integer Programming Problems”. In: (). URL: https://www2.isye.gatech.edu/~ms79/software/ojoc6.pdf.

[24] X.-j. L. Zi-zong Yan and J. Guo. “The existence of a strongly polynomial time simplex algorithm for linear programs”. In: (June 2020). DOI: https://doi.org/10.48550/arXiv.2006.11466.